PolyPublie
Polytechnique Montréal

POLYTECHNIQUE MONTRÉAL
UNIVERSITÉ D'INGÉNIERIE

Titre: Title:	Sécurité des agents mobiles : protocole sécuritaire basé sur un agent sédentaire parfaitement coopérant
Auteur: Author:	Abdelmorhit El Rhazi
Date:	2003
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	El Rhazi, A. (2003). Sécurité des agents mobiles : protocole sécuritaire basé sur un agent sédentaire parfaitement coopérant [Mémoire de maîtrise, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/7005/

Document en libre accès dans PolyPublie
Open Access document in PolyPublie

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/7005/
Directeurs de recherche: Advisors:	
Programme: Program:	Non spécifié

Ce fichier a été téléchargé à partir de PolyPublie, le dépôt institutionnel de Polytechnique Montréal
This file has been downloaded from PolyPublie, the institutional repository of Polytechnique Montréal

https://publications.polymtl.ca

UNIVERSITÉ DE MONTRÉAL

SÉCURITÉ DES AGENTS MOBILES : PROTOCOLE SÉCURITAIRE BASÉ SUR UN AGENT SÉDENTAIRE PARFAITEMENT COOPÉRANT

ABDELMORHIT EL RHAZI
DÉPARTEMENT DE GÉNIE ÉLECTRIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE ÉLECTRIQUE)
Janvier 2003

© Abdelmorhit El Rhazi, 2003

National Library of Canada
Acquisitions and Bibliographic Services
395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada
Acquisitions et services bibliographiques
395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-81514-5

Canada

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé:

SÉCURITÉ DES AGENTS MOBILES : PROTOCOLE SÉCURITAIRE BASÉ SUR UN AGENT SÉDENTAIRE PARFAITEMENT COOPÉRANT

présenté par: Abdelmorhit EL RHAZI

en vue de l'obtention du diplôme de: Maîtrise ès sciences appliquées

a été dûment accepté par le jury d'examen composé de:

M. QUINTERO Alejandro, Ph. D., Président

M. PIERRE Samuel, Ph.D., directeur de recherche

M. BOUCHENEB Hanifa, Ph.D., co-directrice de recherche

M. GUIBAULT François, Ph. D., membre

REMERCIEMENTS

Je désire remercier mon directeur de recherche, le professeur Samuel Pierre, et ma co-directrice, la professeure Hanifa Boucheneb, pour leur patience, leurs conseils et leurs commentaires.

Je désire ensuite remercier tous les membres du Laboratoire de Recherche en Réseautique et Informatique Mobile (LARIM), pour leur aide et leurs critiques.

Je désire également remercier ma famille et mes amis pour leurs encouragements et leur soutien.

RÉSUMÉ

L'utilisation croissante de l'Internet et l'intégration des appareils de l'informatique mobile dans notre vie quotidienne ont suscité un vif intérêt quant aux aspects d'organisation et de communication de l'information à travers des réseaux étendus. Plusieurs architectures ont vu le jour, parmi lesquelles les architectures basées sur les agents mobiles qui se proposent de pallier certains problèmes spécifiques aux architectures client/serveur. L'idée est de remplacer l'envoi des données vers le serveur par la mobilité d'un agent se déplaçant sur l'hôte qui possède les données pertinentes. Cependant, les architectures basées sur cette nouvelle technologie ne sont pas encore déployées à grande échelle dans les applications commerciales. Le principal problème qui freine ce déploiement est lié à l'aspect sécuritaire des agents mobiles et des plates-formes qui les exécutent. En effet, d'une part, les plates-formes, par le fait qu'elles exécutent le code des agents mobiles, sont menacées par leur attaques. D'autre part, lors de son déplacement à travers un réseau, l'agent mobile peut visiter des plates-formes malveillantes et par conséquent il est vulnérable aux nombreuses attaques qui peuvent nuire à son bon fonctionnement.

Ce mémoire a pour but principal de concevoir, de valider formellement et d'implémenter un protocole sécuritaire d'agent mobile qui protège celui-ci contre les attaques des plates-formes malveillantes. Pour y parvenir, nous avons élaboré un protocole sécuritaire protégeant l'agent mobile, et qui est basé sur la coopération parfaite d'un agent sédentaire s'exécutant à l'intérieur d'une tierce plate-forme fiable. Le protocole a pour objectif de protéger une partie du code de l'agent mobile contre la modification et les exécutions incorrectes, de détecter les attaques contre les résultats intermédiaires obtenus à l'intérieur des plates-formes visitées, d'interdire la ré-exécution du code de l'agent mobile, de protéger celui-ci contre l'attaque appelée déni de service et d'enregistrer l'itinéraire de l'agent mobile afin de détecter les attaques liées à sa modification. Nous avons décrit les spécifications du protocole en précisant les tâches affectées à l'agent coopérant ainsi que les scénarios d'attaques que nous avons pu

imaginer et nous avons expliqué la manière dont notre protocole permet de détecter ces attaques.

Après avoir spécifié le protocole, nous l'avons validé formellement afin de vérifier aussi bien ses propriétés générales ainsi que celles liées à la détection des attaques. La validation formelle nous a montré que notre protocole ne comporte aucune erreur de blocage ni de divergence. Nous avons pu expliquer quelques rares erreurs de validation lors de la vérification des propriétés liées à la détection des attaques. Les scénarios qui ont mené à ces erreurs de validation constituaient des comportements concordant avec les scénarios d'attaques prévus dans la phase de la spécification.

Ensuite, nous avons implémenté notre protocole sur une plate-forme d'agents mobiles en prenant comme exemple un agent simple qui visite plusieurs plates-formes, exécute un code critique à l'intérieur de chaque plate-forme avant de retourner chez lui. Les tests de notre implémentation sur les différentes attaques ont révélé que l'agent sédentaire coopérant détecte presque toutes les attaques.

Enfin, dans l'objectif de mesurer les coûts et les performances de notre implémentation, nous avons choisi comme métrique le temps global d'exécution de l'agent mobile et le trafic généré par les différents agents. Nous avons comparé les résultats obtenus avec ceux d'un agent simple qui n'est pas sécuritaire. Les mesures ont démontré que l'ajout de notre protocole à une application d'agents mobiles n'aura qu'une faible influence sur le trafic engendré par l'application elle-même et il n'augmentera que légèrement le temps global d'exécution de celle-ci dans un environnement plus sécuritaire.

ABSTRACT

The huge use of the Internet and the integration of mobile computing devices in our life raise the issue of how the information is organized and communicated through wide-area networks. Several architectures have seen the light, among them the architectures based on the mobile agent concept that offer to solve the specific problems of client/server architecture, such as the number of messages exchanged between clients and servers, which is a stumbling block to the growth of services offered over the internet. However, the deployment of this new architecture in commercial applications requires secure protocols. Indeed, the mobile agent code and data are vulnerable to malicious attacks on the host where the agent is running.

The purpose of this thesis is to design, validate and implement a secure protocol that protects the mobile agent against these attacks. The designed protocol is based on perfect cooperation of a sedentary agent executed on a reliable third party platform. The protocol objective is to protect a part of the mobile agent code against modification and incorrect execution, to detect the attacks against intermediate results obtained on the visited platforms, to deny the mobile agent code re-execution, to protect mobile agent against denial of service attacks and to record the mobile agent itinerary in order to detect its modification. We have described the protocol specifications, especially the behavior of co-operating agents. We have listed the potential attacks and we have described the manner of how the protocol detects them.

A formal verification method allows us to validate the protocol specifications. We have constructed the formal model, simulated it and validated the general features and those related to the attack detection. The formal validation has demonstrated that the protocol contains neither deadlock nor livelock.

Finally, the protocol has been implemented on Grasshopper platform that allowed us to measure the effectiveness and the cost of our algorithms. To test the protocol, we chose as example a simple agent that visits several platforms, runs critical

code on each platform and turns back. The tests revealed the co-operating agent detects the majority of attacks. We have measured the execution time and the traffic generated by the agents. We concluded that the add of our protocol to a mobile agent application affects only slightly the network load and the global execution time of the application.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES ANNEXES

CHAPITRE I
INTRODUCTION

L'utilisation sans cesse croissante de l'Internet et l'intégration des appareils de l'informatique mobile dans notre vie quotidienne ont suscité un vif intérêt quant aux aspects d'organisation et de communication de l'information à travers des réseaux étendus. Plusieurs architectures ont vu le jour. La plus connue et la plus utilisée est l'architecture client/serveur caractérisée par les échanges de messages de type requête/réponse entre les clients et les serveurs par l'intermédiaire de réseaux. Les architectures basées sur les agents mobiles se proposent de pallier certains problèmes spécifiques aux architectures client/serveur comme le nombre accru de messages échangés entre les clients et les serveurs, problème généralement à la base d'un ralentissement des services offerts via Internet. Cependant, le déploiement de ces nouvelles architectures dans des applications commerciales nécessite des protocoles sécuritaires, objets du présent mémoire. Dans ce chapitre, nous allons d'abord introduire les concepts de base, puis nous définirons les éléments de la problématique, suivis d'un exposé des objectifs de recherche et d'une esquisse méthodologique, pour enfin présenter le plan de notre mémoire.

1.1 Définitions et concepts de base

Le terme *agent mobile* contient deux concepts différents : agent et mobilité. Un *agent* est un programme indépendant, souvent implémenté comme des processus légers (thread) multiples, qui possède les propriétés suivantes : autonomie, sociabilité, réactivité et pro-activité. L'autonomie permet à l'agent d'opérer sans l'intervention directe humaine ou autre en possédant un certain contrôle de ses actions et de son état interne. La *sociabilité* est le fait que l'agent interagit avec d'autres agents en utilisant un langage de communication agent. La réactivité permet à l'agent de percevoir son environnement (qui peut être le monde physique, l'utilisateur à travers son interface

graphique, une collection d'autres agents, Internet, et peut être tout cela combiné) et de répondre au changement qui survient. Enfin, la pro-activité donne à l'agent l'habilité de prendre l'initiative pour accomplir ses objectifs. Un agent est *mobile* lorsqu'il peut être transporté d'une machine à une autre dans un réseau. Un agent mobile ne se transporte pas tout seul, c'est la plate-forme où il s'exécute qui se charge de son envoi vers une autre plate-forme.

En comparant le paradigme d'agent mobile avec son rival client/serveur, les systèmes d'agents mobiles possèdent de multiples avantages. En effet, en migrant à l'endroit où réside la ressource voulue, l'agent interagit avec cette ressource sans transmettre des données à travers le réseau, cela réduit d'une manière significative la consommation de la bande passante. De la même manière, en migrant à l'endroit où réside l'utilisateur, l'agent peut répondre rapidement aux actions de l'utilisateur. Dans ces cas, l'agent peut continuer son interaction avec la ressource ou l'utilisateur, même si la connexion réseau est rompue. Cette caractéristique rend le concept d'agent particulièrement attractif pour les applications de l'informatique mobile. De plus, la majorité des applications réparties concordent naturellement avec le modèle des agents mobiles, puisqu'un agent peut migrer à travers un ensemble de machines, envoyer des agents fils pour visiter parallèlement de multiples machines, rester stationnaire et interagir avec des ressources distantes, ou bien combiner toutes ces tâches à la fois. Finalement, la mobilité de l'agent cache aux développeurs les spécifications de réseaux et leur fournit un paradigme familier de développement.

Les champs d'application des systèmes d'agents mobiles sont très variés. Les plus connus sont les applications de commerce électronique ou e-commerce, de distribution de logiciels, de recherche d'informations, d'administration de systèmes et de gestion de réseaux. Mais, le déploiement à grande échelle des agents mobiles est freiné par le manque d'applications commerciales répandues, les impératifs de la sécurité qui n'a pas encore atteint un niveau acceptable de maturité et le manque d'un standard bien éprouvé pour supporter les développeurs d'applications mobiles.

1.2 Éléments de la problématique

Parmi les raisons qui empêchent les concepteurs et les développeurs d'adopter le paradigme d'agent mobile dans le développement de leurs systèmes répartis, il convient de mentionner l'aspect sécurité de ce type de technologie qui n'a pas encore atteint un degré de maturité suffisant. Généralement, les systèmes basés sur les agents mobiles subdivisent le problème de la sécurité en deux volets différents. Le premier concerne la protection de la plate-forme qui exécute l'agent contre toute attaque visant à nuire aux ressources de la plate-forme. Cet aspect de la sécurité est assez bien résolu par les abondants travaux qui ont été réalisés lors de l'apparition du code mobile (Applet), les virus et les vers à travers l'Internet [WAL99] [KAR00] [ZHO98]. Le second problème de la sécurité concerne la protection de l'agent mobile contre les attaques des plates-formes malveillantes. L'importance de ce type de protection vient du fait que l'agent mobile peut visiter dans son itinéraire des plates-formes qui ne sont pas nécessairement fiables. Par conséquent, elles peuvent modifier le code et/ou les données de l'agent mobile, le ré-exécuter afin d'obtenir des résultats non permis ou tout simplement le détruire et retarder son exécution. D'où la nécessité de doter les agents mobiles des mécanismes qui le protègent contre ces attaques.

Les travaux de recherche actuels [ALG00] [HOH98] [SAN98] [HOH00] ont concentré leurs efforts sur la protection et la détection des attaques visant le code et les données de l'agent mobile. Certaines de ces approches utilisent les notions de la cryptographie pour masquer le code et les données de l'agent mobile à l'intérieur des plates-formes. Les autres utilisent des états de référence et des délais de protection pour détecter ou protéger l'agent contre les attaques. Mais aucune de ces approches n'a pu concevoir un protocole complètement sécuritaire qui traite toutes les différentes attaques possibles.

Parmi les attaques qui n'ont pas encore fait l'objet de travaux ni de protection ni de détection figure l'attaque qui consiste à ré-exécuter le code de l'agent mobile ou une partie de son code afin que la plate-forme attaquante puisse obtenir des privilèges illégaux ou des résultats biaisés comme la ré-exécution d'un ordre d'achat ou la ré-

exécution de l'agent avec des données différentes. Une seconde attaque qui n'a pas suscité l'intérêt des travaux de recherche est l'attaque appelée *déni de service* qui consiste à tuer l'agent mobile ou retarder son exécution. Imaginons un agent mobile de magasinage électronique qui porte une offre dont le délai de validité est court, la plate-forme attaquante peut retarder l'exécution de l'agent jusqu'à l'expiration de cette offre. Quant à la protection de l'agent mobile contre la manipulation de ses interactions avec les autres agents ou la plate-forme, elle n'est pas encore assurée.

L'importance de doter l'agent mobile de mécanismes qui le protègent contre ces attaques est incontestable, surtout pour les applications spécifiques au domaine du commerce électronique où la sécurité est cruciale. En effet, ces attaques (ré-exécution de l'agent, déni de service, ...) ont une influence directe sur le degré de fiabilité des systèmes utilisant la technologie agent mobile dans le domaine du e-commerce ou du m-commerce (*mobile electronic commerce*).

1.3 Objectifs de recherche

L'objectif principal de ce mémoire est de concevoir, de valider et d'implémenter un protocole sécuritaire d'agent mobile qui protège celui-ci contre les attaques des plates-formes malveillantes. Plus spécifiquement, ce mémoire vise à :

- concevoir et spécifier un protocole sécuritaire basé sur un agent sédentaire parfaitement coopérant avec notre agent mobile en définissant particulièrement les tâches affectées à cet agent ;
- valider formellement et implémenter les spécifications de notre protocole sur une plate-forme d'agents mobiles afin de s'assurer que le protocole protège l'agent mobile contre les attaques ;
- évaluer l'efficacité de notre protocole à travers une étude de cas.

1.4 Esquisse méthodologique

L'idée générale de notre approche est d'associer à l'agent mobile un agent dit *sédentaire* qui se chargera de protéger ou de détecter les éventuelles attaques qui altèrent le bon fonctionnement de l'agent mobile. En particulier, l'agent sédentaire aura comme rôle la détection des exécutions incorrectes d'une partie de code de l'agent, la protection de l'agent mobile contre la ré-exécution de son code, contre le déni de service et contre la modification de son itinéraire.

Le modèle de notre système est le suivant. L'agent mobile se déplace d'une plate-forme à une autre qui n'est pas nécessairement fiable. Dans chaque plate-forme visitée, notre agent mobile établit une connexion fiable avec un agent sédentaire qui lui est dédié. L'agent sédentaire est lancé au même moment que notre agent mobile. Une tierce plate-forme fiable exécute l'agent sédentaire et elle doit être disponible à tout moment. L'agent sédentaire coopère avec l'agent mobile pour assurer la sécurité de celui-ci. Cette coopération est dite *parfaite* vu que les autres approches qui ont utilisé la coopération des agents l'ont restreint à l'enregistrement de l'itinéraire de l'agent mobile [VOL98] [ALL01].

Les spécifications seront développées en imaginant des scénarios d'attaques et en réfléchissant sur les éventuelles solutions à l'aide de la coopération de l'agent sédentaire sur lequel repose la sécurité de notre système. L'implémentation de ces spécifications pourra se faire sur la plate-forme Grasshopper ou sur une autre plate-forme dont le code source est libre. Afin de valider formellement notre protocole, nous serons amenés à modéliser celui-ci et à écrire ce modèle à l'aide d'un langage formel (Promela en l'occurrence). L'environnement SPIN nous permettra de simuler le modèle et de vérifier certaines de ses propriétés. Pour tester notre protocole, nous pouvons utiliser les scénarios imaginés dans la partie des spécifications. La mesure de l'efficacité et du coût de notre protocole prendra comme métrique le temps de réponse, le degré de complexité des algorithmes et le nombre de messages échangés dans le réseau.

1.5 Plan du mémoire

Le chapitre 2 expose la problématique de la sécurité dans les systèmes basés sur le paradigme d'agent mobile et les principales approches de résolution proposées dans la littérature. Le chapitre 3 décrit les tâches affectées à l'agent sédentaire et la communication entre celui-ci et l'agent mobile ; il explique aussi la manière dont notre protocole protège l'agent mobile contre les différentes attaques. Le chapitre 4 présente le modèle formel de notre protocole, ainsi que son implémentation en précisant et en analysant les résultats obtenus. En guise de conclusion, le chapitre 5 résume les principaux résultats que nous avons obtenus, fait état des limitations de l'implémentation du protocole et donne des indications en vue de recherches futures.

CHAPITRE II
LA SÉCURITÉ DES AGENTS MOBILES

Les architectures basées sur les agents mobiles ne sont pas encore déployées à grande échelle dans les applications commerciales. Le principal problème qui heurte ce déploiement est lié à l'aspect sécuritaire des agents mobiles et des plates-formes qui les exécutent. En effet, d'une part, les plates-formes, par le fait qu'elles exécutent le code des agents mobiles, sont menacées par leurs attaques. D'autre part, lors de son déplacement à travers un réseau, l'agent mobile peut visiter des plates-formes malveillantes et par conséquent il est vulnérable aux nombreuses attaques qui peuvent nuire à son bon fonctionnement. De nombreux travaux de recherche ont vu le jour traitant l'aspect sécuritaire des agents mobiles, mais aucun d'eux n'a réussi à concevoir un système complètement sécuritaire. Dans ce chapitre, nous allons tout d'abord décrire les différentes attaques menaçant l'agent mobile. Puis, nous proposerons une manière de classer ces attaques par ordre d'importance. Ensuite, nous expliquerons les principales approches actuelles qui ont traité la sécurité des agents mobiles, pour enfin conclure avec une comparaison des approches.

2.1 Les attaques des plates-formes malveillantes

La sécurité dans un système d'agents mobiles couvre, selon Walsh *et al.* [WAL99], quatre aspects différents :

- la sécurité de la transmission de l'agent ;
- la protection de la plate-forme contre les agents malveillants ;
- la protection de l'agent contre l'attaque d'un autre agent malveillant ;
- et la protection de l'agent contre l'attaque d'une plate-forme malveillante.

La plate-forme qui exécute les agents mobiles est vulnérable à plusieurs attaques dues à l'exécution du code mobile. Ceci inclut la lecture non permise des informations sensibles de la plate-forme, le dommage de ses ressources (CPU, mémoire ...) et l'attaque dite *déni de service* i.e. arrêt partiel ou complet des services offerts par la plate-forme. En même temps, les agents qui sont fiables doivent avoir accès aux ressources de la plate-forme pour qu'ils puissent s'exécuter normalement. Pour se faire, le système doit fournir à la plate-forme un mécanisme d'authentification qui lui permet de spécifier les droits d'accès aux agents. Le problème [ALG00] de la protection de la plate-forme contre les attaques des agents mobiles malveillants a reçu de considérables attentions de la part des travaux de recherches justifiées par les imminentes menaces des virus, des vers[1] et des chevaux de Troie[2].

Les agents mobiles, de leur part, sont vulnérables à plusieurs types d'attaques des plates-formes malveillantes où ils exécutent leur codes. Ceci inclut les attaques passives qui ne manipulent ni les données ni le code de l'agent mobile comme l'analyse du trafic, et les attaques actives qui modifient les données et le code de l'agent mobile. Les attaques passives sont difficiles à détecter, mais les agents mobiles peuvent être protégés contre elles en appliquant des mécanismes cryptographiques. Par contre, les attaques actives sont relativement plus faciles à détecter, mais il est plus difficile de protéger les agents mobiles contre leurs effets.

2.1.1 Description des attaques d'une plate-forme malveillante

F. Hohl [HOH98] définit une plate-forme malveillante comme étant une plate-forme capable d'exécuter un agent provenant d'une autre plate-forme, et qui essaye de nuire au bon fonctionnement de l'agent mobile d'une quelconque manière. On a identifié [HOH98] les différentes attaques d'une plate-forme contre un agent mobile suivantes :

1. Espionnage du code ;

---

[1] Un ver est un programme qui peut s'auto-reproduire et se déplacer à travers un réseau en utilisant les mécanismes réseau.

[2] On appelle "Cheval de Troie" (an anglais *trojan horse*) un programme informatique effectuant des opérations malicieuses à l'insu de l'utilisateur. Le nom "Cheval de Troie" provient d'une légende narrée dans l'*Iliade* (de l'écrivain *Homère*) à propos du siège de la ville de Troie par les Grecs.

2. Espionnage des données ;
3. Espionnage de l'état ;
4. Manipulation du code ;
5. Manipulation des données ;
6. Manipulation de l'état ;
7. Exécution incorrecte du code ;
8. Mascarade de la plate-forme ;
9. Déni de service ;
10. Espionnage des interactions avec les autres agents ;
11. Manipulation des interactions avec les autres agents ;
12. Retour des résultats erronés des appels systèmes effectués par l'agent ;
13. Ré-exécution de l'agent.

1-Espionnage du code

Pour exécuter l'agent mobile, la plate-forme doit être en mesure de lire son code. La connaissance du code de l'agent mobile mène à :

- la connaissance de la stratégie d'exécution de l'agent ;
- la connaissance des structures physiques du code et des données dans la mémoire de la plate-forme ;
- et parfois à la connaissance d'une partie des données de l'agent mobile.

2-Espionnage des données

La lecture des données privées d'un agent mobile par une plate-forme est très critique vu que cette attaque ne laisse pas de trace contrairement à l'attaque de modification des données. C'est un problème qui concerne certaines classes de données dont la simple connaissance implique une perte de confidentialité. La clé privée est un exemple de cette catégorie de données.

3-Espionnage de l'état

Dés que la plate-forme prend connaissance du code de l'agent mobile et ses données, elle peut déterminer l'étape suivante de son exécution. Et même si on protège les données utilisées par l'agent mobile, il est difficile de protéger l'information concernant son état d'exécution. C'est un problème important parce qu'avec la connaissance du code une plate-forme malveillante peut déduire des informations sur l'état de l'agent. Par exemple, elle peut savoir si une offre est meilleure ou non en observant simplement l'état d'exécution sans connaître les données de l'agent mobile.

4-Manipulation du code

Si la plate-forme est capable de lire le code et si elle peut accéder à la mémoire réservée au code, elle aura l'habilité de le modifier. Cette modification peut être d'une manière permanente en implémentant des virus, des vers et des chevaux de Troie afin d'attaquer les autres plates-formes se trouvant dans l'itinéraire de l'agent mobile; ou bien temporaire en modifiant seulement la manière selon laquelle la plate-forme exécute l'agent mobile.

5-Manipulation des données

Si la plate-forme connaît la localisation physique des données dans la mémoire et la sémantique de leurs éléments, elle peut modifier ces données. Une plate-forme attaquante peut modifier aussi bien les résultats obtenus par l'agent mobile à l'intérieur des plates-formes précédentes, que les données propres à l'agent mobile (l'identité de la plate-forme d'origine, le prix d'un produit, les caractéristiques techniques ...).

6-Manipulation de l'état

Même si on parvient à sécuriser les données de l'agent mobile, une plate-forme malveillante peut effectuer une attaque en manipulant l'état de l'agent. Elle peut conduire le comportement de celui-ci en modifiant son état d'exécution. La plate-forme peut par exemple forcer l'agent à faire un choix qui avantage la plate-forme attaquante.

7-Exécution incorrecte du code

Sans changer ni le code ni l'état d'exécution de l'agent mobile, une plate-forme malveillante peut modifier la façon dont elle exécute le code de l'agent. Elle peut par exemple retourner une fausse valeur quand elle procède à la comparaison de son prix avec un prix nominal fixé par la plate-forme d'origine.

8-Mascarade de la plate-forme

La plate-forme qui envoie l'agent mobile est responsable de s'assurer de l'identité de la plate-forme qui reçoit cet agent. Si une tierce partie intercepte l'agent ou fait une copie de celui-ci, elle pourra masquer son identité en passant par celle du vrai receveur et commencer l'exécution de l'agent. La mascarade est suivie généralement d'une autre attaque comme l'attaque de l'espionnage des données.

9-Déni de service

Une plate-forme malveillante peut simplement s'abstenir d'exécuter l'agent mobile i.e. arrêter ou retarder son exécution. Cette action est considérée comme une attaque vu que cette plate-forme peut connaître le dernier délai de la validité de l'offre d'une autre plate-forme. La plate-forme attaquante peut retarder l'agent jusqu'à ce que le délai de l'offre soit expiré.

10- Espionnage des interactions avec les autres agents

Un agent mobile peut effectuer des achats à distance dans une plate-forme autre que celle où il est entrain d'exécuter son code. Si les interactions entre l'agent et cette plate-forme ne sont pas protégées, la plate-forme courante peut observer ces achats même s'elle ne peut espionner l'exécution du code de l'agent mobile. Cette attaque peut mener à l'attaque de la manipulation de ces interactions.

11-Manipulation des interactions avec les autres agents

Si la plate-forme courante peut manipuler les interactions de l'agent mobile avec les agents qui s'exécutent à l'intérieur des autres plates-formes, elle peut agir avec l'identité de cet agent. La plate-forme attaquante peut ainsi rediriger ces interactions de l'agent sur une autre plate-forme.

12-Retour des résultats erronés des appels systèmes effectués par l'agent

Une plate-forme attaquante peut retourner des résultats erronés des appels systèmes que l'agent mobile exécute. À titre d'exemple, lorsque l'agent mobile veut vérifier si la plate-forme courante est bien sa plate-forme d'origine, il fait un appel système pour connaître l'identité de la plate-forme courante. Celle-ci peut retourner une fausse adresse en se faisant passer par la plate-forme d'origine. L'agent se trompe et agit comme s'il était chez lui.

13-Ré-exécution de l'agent

La ré-exécution a lieu lorsqu'une plate-forme copie l'agent, une partie de l'agent ou un message de celui-ci et le ré-exécute. Par exemple, une plate-forme malveillante peut ré-exécuter un message d'achat d'un produit ou de payement d'une facture, ou tout simplement ré-exécuter l'agent mobile avec des données différentes.

La Figure 2.1 récapitule les différentes attaques possibles d'une plate-forme malveillante contre un agent mobile.

2.1.2 Classification des attaques par ordre d'importance

L'importance d'une attaque par rapport aux autres dépend du type d'application qui implémente l'agent mobile. L'attaque qui consiste à ré-exécuter l'agent mobile, par exemple, est très importante dans le cas d'une application qui fait intervenir des opérations idempotentes i.e. opérations dont la ré-exécution influence l'intégrité du système, comme l'opération d'achat d'un article ou du payement d'une facture. Cette même attaque n'a pas d'importance dans une application de recherche d'information où

la ré-exécution de l'agent mène aux même résultats. De ce fait et afin d'ordonner les attaques selon leur importance, nous les avons classées dans trois catégories. Le Tableau 2.1 récapitule les différentes catégories des attaques classées par ordre d'importance.

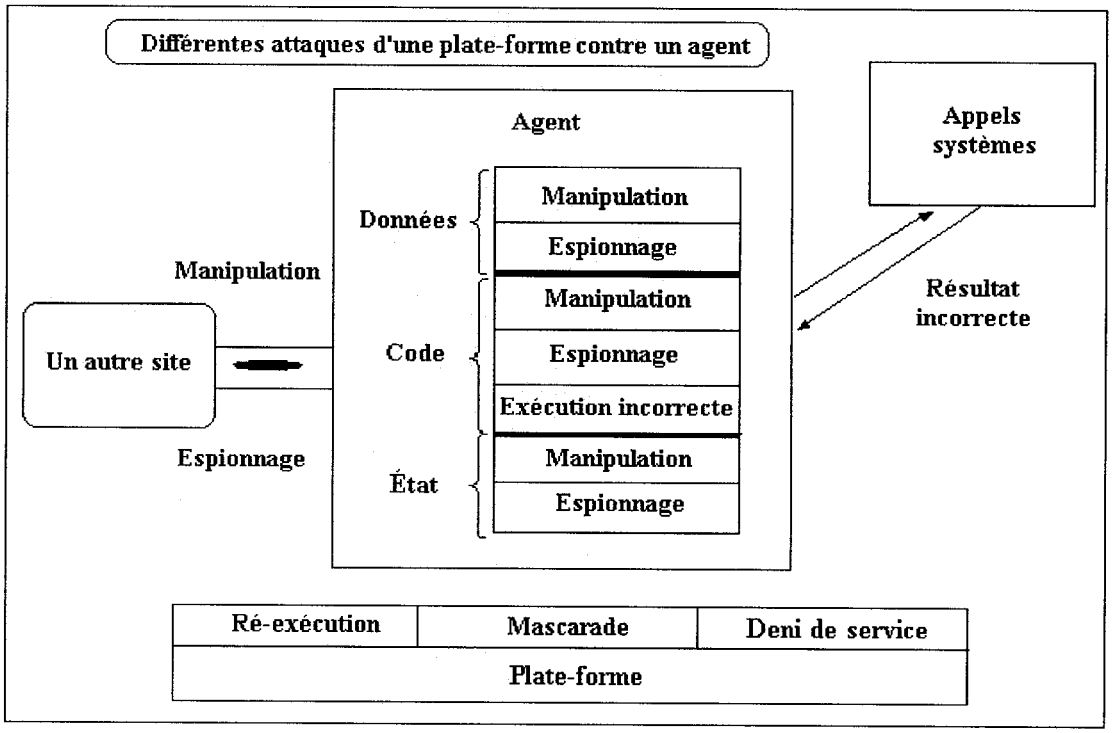

Figure 2.1 Différentes attaques d'une plate-forme contre l'agent mobile

La première catégorie rassemble les attaques qui concernent tous les types d'application. Cette catégorie a une très grande importance vu qu'un système d'agents mobiles n'est considéré comme sécuritaire que s'il fournit des mécanismes de protections ou au moins de détection de ces attaques. Cette catégorie contient les attaques suivantes : manipulation du code, manipulation des données, manipulation de l'état, exécution incorrecte du code et retour des résultats erronés des appels systèmes effectués par l'agent.

Tableau 2.1 Catégorie des attaques

Catégorie	Attaques
Première catégorie : Attaques concernant toutes les applications	Manipulation du code Manipulation des données Manipulation de l'état Exécution incorrecte du code Retour des résultats erronés des appels systèmes effectués par l'agent
Deuxième catégorie : Attaques concernant des applications spécifiques	Déni de service Manipulation des interactions avec les autres agents Ré-exécution de l'agent
Troisième catégorie : Attaques menant à d'autres attaques	Espionnage du code Espionnage des données Espionnage de l'état Mascarade de la plate-forme Espionnage des interactions avec les autres agents

La deuxième catégorie est constituée des attaques qui ne concernent que des applications spécifiques. Les systèmes d'agents mobiles lui donnent moins d'importance que la première catégorie vu le nombre limité des applications qui devront implémenter des mécanismes de protection ou de détection de ces attaques pour leur bon fonctionnement. Les attaques qui font partie de cette catégorie sont : le déni de service qui a une très grande importance pour les applications qui sont sensibles au délai d'exécution de l'agent mobile, la manipulation des interactions avec les autres agents qui concerne les applications qui font intervenir ces types d'interactions et la ré-exécution de l'agent qui a une grande importance dans le champs des applications qui exécutent des opérations idempotentes.

La dernière catégorie rassemble les attaques qui mènent à d'autres attaques de la première ou la deuxième catégorie. Les attaques ne constituent pas une finalité en soit mais un passage vers les autres attaques. Une plate-forme qui attaque l'agent mobile avec une mascarade cherche à effectuer une autre attaque de manipulation du code, par exemple. L'importance de cette catégorie est faible vu que si l'agent mobile est protégé contre les attaques de la première et/ou de la deuxième catégorie, les attaques de la dernière catégorie n'auront aucun effet. Les attaques qui peuvent être classées dans cette catégorie sont : espionnage du code, espionnage des données, espionnage de l'état, mascarade de la plate-forme et espionnage des interactions avec les autres agents.

2.2 Protection des agents mobiles contre les attaques

Alors que les contre-mesures visant à protéger les plates-formes sont largement inspirées des systèmes conventionnels en employant des méthodes préventives [ALL01], les techniques de protection des agents font plutôt de la détection. Cela est dû au fait que l'agent est complètement dépendant de la plate-forme sur laquelle il s'exécute et ne peut, par lui-même, empêcher une attaque. Par contre, il est possible de la détecter ou de la rendre moins dangereuse.

La protection des agents mobiles contre la plate-forme est un domaine de recherche très important car les problèmes qui se posent sont différents de ceux rencontrés dans les systèmes traditionnels. Parmi les techniques de protection les plus importantes, on trouve :

1. la protection cryptographique des agents mobiles (Sander et Tschudin, 1998),
2. la sécurité cryptographique d'un code mobile (Algesheimer *et al.*, 2000),
3. la sécurité cryptographique des données (Karnik et Tripathi, 2000),
4. la protection des agents mobiles en utilisant les états de référence (Vigna, 1998; Farmer *et al.*, 1996; Minsky *et al.*, 1996)
5. la boîte noire sécuritaire limitée dans le temps (Fritz Hohl, 1998),
6. la protection mutuelle des agents coopérants (Roth, 1998; AlléeAllée, 2001).

2.2.1 Protection cryptographique des agents mobiles

Sander *et al.* [SAN98] ont conçu une approche qui garantie la confidentialité des calculs des agents mobiles. Ils ont fixé comme objectif de pallier aux trois problèmes fondamentaux que rencontre l'exécution de l'agent mobile dans un environnement non fiable :

- L'agent mobile peut-il s'auto-protéger contre les modifications frauduleuses d'une plate-forme malveillante ? (l'intégrité du code et de son exécution)
- L'agent mobile peut-il dissimuler son programme qu'il veut exécuter ? (confidentialité du code)
- L'agent mobile peut-il signer un document distant sans révéler la clé privée de l'utilisateur ? (calcul avec des secrets en public)

Les solutions cryptographiques de la sécurité d'exécution des agents mobiles ne doivent pas utiliser des protocoles interactifs i.e. impliquant le créateur de l'agent. Le but escompté est de rendre l'agent mobile capable d'exécuter des calculs sensibles d'une manière sécuritaire même si cette exécution se fait à l'intérieur d'une plate-forme non fiable.

Les remarques suivantes s'appliquent à un agent mobile :

- les données en claire peuvent être lues et modifiées ;
- le code en claire peut être manipulé ;
- les messages en claires peuvent être falsifiés.

L'idée de base de cette approche est fondée sur le fait qu'il n'existe aucune raison intrinsèque pour qu'un programme soit exécuté dans une forme claire. Dans le même sens qu'on peut communiquer des messages cryptés sans les comprendre, on veut qu'un ordinateur exécute un programme crypté sans qu'il le comprenne. Si on peut exécuter un programme crypté sans le décrypter, on pourra assurer la confidentialité du code et l'intégrité du celui-ci dans le sens où la modification frauduleuse ne sera pas possible. Par conséquent, les attaques des plates-formes malveillantes seront réduites au déni de service, aux modifications aléatoires du code et à la ré-exécution de celui-ci.

L'objectif de cette approche est de crypter des fonctions de manière à ce qu'on puisse implémenter leurs transformations dans un programme. Le programme résultant est constitué des instructions en claire qu'un processeur ou un interpréteur comprend. Ce que le processeur ne va pas comprendre est le sens des fonctions initiales. La Figure 2.2 illustre le problème que Sander *et al.* veulent résoudre.

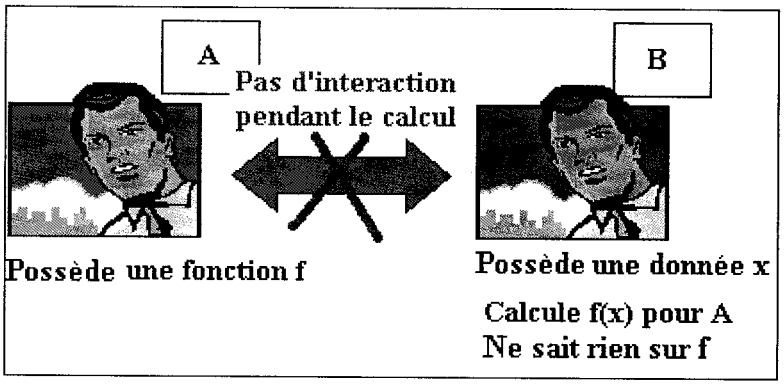

Figure 2.2 Problème du protocole de Sander

On suppose que la fonction initiale peut être transformée (cryptée) à une autre fonction *E(f)*. On note *P(f)* le programme qui implémente la fonction *f*. Le protocole est illustré dans la Figure 2.3.

Le défi de cette approche est de trouver une méthode cryptographique pour une fonction quelconque. Sander *et al.* [SAN98] identifient des classes de fonctions pour lesquelles on peut trouver des transformations cryptographiques. Une classe intéressante est la classe des fonctions polynomiales et rationnelles. Mais même pour cette classe restrictive, ils n'ont pas pu trouver des solutions complètes pour toutes les fonctions polynomiales. Ils n'ont pas trouvé une solution générale pour protéger l'agent mobile. Cependant, ils ont mis en évidence qu'un code peut être partiellement protégé contre les plates-formes malveillantes.

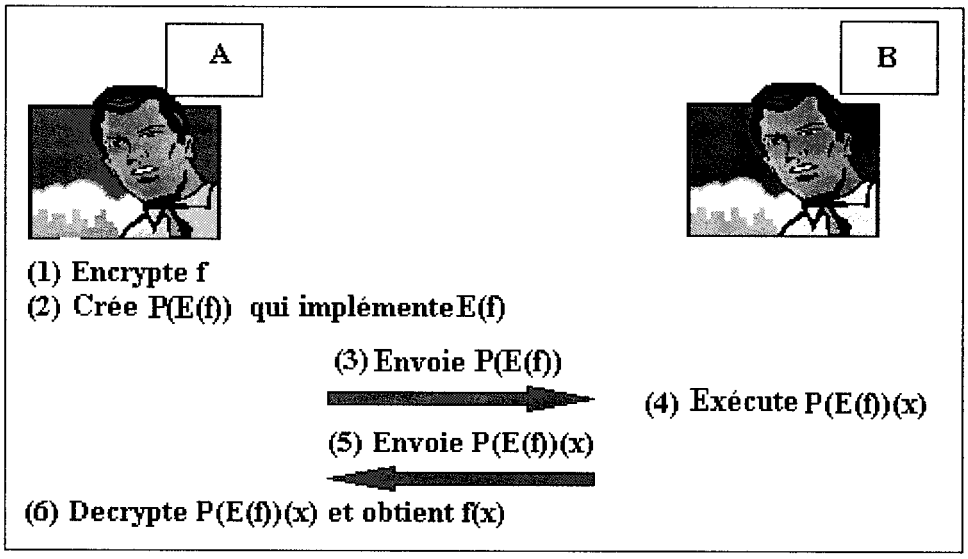

Figure 2.3 Protocole des fonctions cryptographiques

2.2.2 Sécurité cryptographique d'un code mobile

La protection du code mobile a été considérée comme impossible par des chercheurs dans ce domaine, jusqu'à ce que Sander et Tschudin [SAN98] réalise qu'avec les protocoles basés sur la théorie de la cryptographie on peut exécuter une forme encryptée du code mobile dans une plate-forme malveillante. Sander et Tschudin ont conclu que seules les fonctions qui sont représentées dans une forme polynomiale peuvent s'exécuter d'une manière sécuritaire. Mais, le défaut de leur approche est qu'aucune information ne doit être révélée à la plate-forme visitée et que seule la plate-forme d'origine a le droit d'avoir des résultats d'exécution. Ceci a pour effet de limiter le comportement de l'agent à l'intérieur d'une plate-forme.

La seule protection existante pour un agent mobile contre les attaques d'une plate-forme malveillante utilise un matériel fiable supplémentaire. Ceci a été proposé par Yee [YEE99] et par Wilhelm *et al.* [WIL99] et exige l'exécution du code mobile exclusivement dans un matériel inviolable, et l'encrypte aussitôt qu'il quitte ce matériel. La solution proposée par Yee [YEE99] et par Wilhelm *et al.* [WIL99] utilise un module matériel fiable dans chaque plate-forme participante, comme des cartes à puce ou des

co-processeurs cryptographiques. Chaque module matériel possède une clé publique. Un code mobile peut être exécuté d'une manière sécuritaire dans cette infrastructure de la façon suivante. Après la génération du code de l'agent mobile, sa plate-forme d'origine l'encrypte avec la clé publique de la première plate-forme dans l'itinéraire de l'agent. Dès la réception du code par une plate-forme, elle l'envoie au module matériel avec ses paramètres. Le module décrypte le code, l'exécute avec les entrées fournies et l'encrypte ensuite avec la clé publique de la plate-forme suivante. Puis, il retourne les sorties du son exécution en claire à la plate-forme qui se charge d'envoyer le code crypté à la plate-forme suivante.

Algesheimer *et al.* [ALG00] ont conçu une architecture pour l'exécution sécuritaire d'un fragment du code mobile qui ne nécessite pas un matériel additionnel. Leur approche est basée sur l'implémentation d'une tierce partie fiable et minimale. Celle-ci effectue un calcul sécuritaire en exécutant des opérations cryptographiques à la place de la plate-forme courante. Elle assure la confidentialité et l'intégrité des calculs. En plus que cette tierce partie ne doit savoir rien sur le sens des opérations calculées, elle doit être fiable et ne collabore ni avec la plate-forme d'origine ni avec une autre plate-forme pour attaquer l'agent mobile. Elle doit être toujours en-ligne et connectée à toutes les plates-formes constituant l'itinéraire de l'agent. Elle exécute les applications agents et est à leurs dispositions pour effectuer des calculs sécuritaires. La Figure 2.4 illustre le principe de cette approche.

La technique de cette approche est basée sur le chiffrement d'un circuit digital binaire qui effectue le calcul de la partie de l'agent qui nécessite la confidentialité. En principe, cette technique est conseillée pour des petites parties de l'agent, comme la comparaison des offres des plates-formes dans des agents mobiles de magasinage électronique.

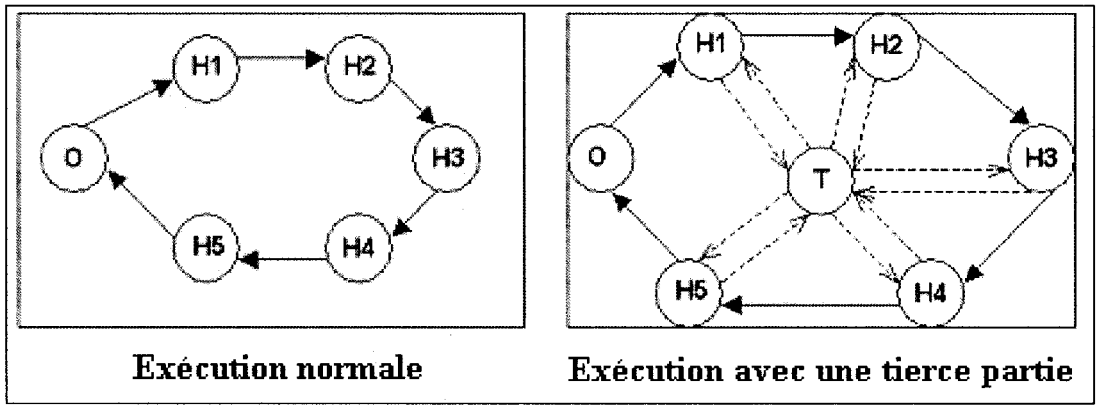

Figure 2.4 Protocole sécuritaire basé sur une tierce partie

2.2.3 Sécurité cryptographique des données

Karnik *et al.* [KAR00] ont conçu un système d'agents mobiles, appelé *Ajanta,* qui a pour objectif d'adresser les problèmes de la sécurité des données des agents. Chaque agent possède un certificat inviolable, appelé sa *carte d'identité*, assignée par son créateur. Elle contient le nom de l'agent et les noms de son propriétaire et de son créateur. Ce certificat est signée par le propriétaire de l'agent mobile. Elle contient aussi l'URL du serveur qui fournit les classes requises par l'exécution de l'agent à l'intérieur d'une plate-forme distante.

Afin de détecter les attaques des plates-formes malveillantes contre l'agent mobile, le système Ajanta a doté celui-ci de trois mécanismes agissant sur ses données et ses états. Le premier mécanisme consiste à déclarer une partie des données de l'agent mobile comme étant en lecture seule (*read only*). Cette partie contient des données demeurant constantes le long du trajet de l'agent mobile. Une plate-forme ne peut modifier le contenu de cette partie de données sans que la plate-forme d'origine ne détecte cette attaque. Le second mécanisme permet à l'agent mobile de créer un journal en ajout seul (*append only*) où une plate-forme visitée ne peut insérer qu'un seul enregistrement contenant ses résultats. Le dernier mécanisme, appelé *révélation sélective*, permet à la plate-forme d'origine de designer un certain nombre de plates-formes dans lesquelles une partie des données de l'agent mobile est visible.

État en lecture seule

Souvent, l'agent mobile contient des éléments constants comme faisant partie de son état et de ses données. Par exemple, sa carte d'identité ne peut être modifiée que par sa plate-forme d'origine et elle doit demeurer constante durant le cycle de vie de l'agent mobile. Le système Ajanta fournit un mécanisme cryptographique pour protéger de telles données constantes. La classe agent générique fournie par Ajanta contient un objet *ReadOnlyContainer*. Ce dernier contient un vecteur d'objet de différents types signé par la plate-forme d'origine de l'agent mobile. Durant, la création de l'agent, ce vecteur est initialisé avec la valeur des données constantes de l'agent mobile. La signature est calculée en appliquant une fonction de hashage à ce vecteur, puis on crypte le résultat avec la clé privée de la plate-forme d'origine.

(Signature = Encrypt(Hash(Vecteur), Clé privée de la plate-forme d'origine)).

Ce calcul doit être fait à l'intérieur de la plate-forme d'origine de l'agent mobile. Le *ReadOnlyContainer* contient une méthode *Verify* qui permet à chaque plate-forme dans l'itinéraire de l'agent mobile de vérifier si jamais la partie constante a été frauduleusement modifiée par une plate-forme précédente. Pour se faire, les plates-formes ont accès à la clé publique de la plate-forme d'origine. Si une plate-forme malveillante modifie la partie en lecture seule, la signature de la plate-forme d'origine ne sera plus valide. En plus qu'elle ne peut recréer cette partie fixe vu qu'elle ne possède pas la clé privée de la plate-forme d'origine.

Journal en ajout seul

Dans certaines situations, l'agent mobile collecte des données à partir des plates-formes visitées, et veut protéger ces données contre la modification frauduleuse. Pour se faire, l'agent crée un journal en ajout seul qui fait parti de son état. Une plate-forme visitée ne peut qu'ajouter des éléments dans ce journal. Elle ne peut ni supprimer ni modifier les enregistrements de celui-ci. Si l'agent mobile a besoin de la confidentialité des données, les éléments sont cryptés avec la clé publique de la plate-forme d'origine

avant leur ajout. L'agent mobile utilise un vecteur qui contient les objets à protéger, leurs signatures électroniques et les identités des plates-formes signataires. Il contient aussi une somme de contrôle (*checksum*) utilisée pour détecter les modifications frauduleuses. Quand l'agent mobile est créé, ce vecteur est vide. La somme de contrôle est initialisée en cryptant un nombre aléatoire (*nonce*) avec la clé publique de la plate-forme d'origine de la manière suivante:

(checksum = Encrypt(N_a, Clé publique de la plate-forme d'origine)).

Ce nombre N_a n'est connu que par la plate-forme d'origine de l'agent mobile et il doit rester secret.

Durant son parcourt, l'agent mobile peut insérer un objet (X) dans ce vecteur. Il demande à la plate-forme courante (P) de signer cet objet en utilisant sa propre clé privée. L'objet, sa signature et l'identité de la plate-forme sont insérés dans le vecteur. Puis, la somme de contrôle est mise à jour de la manière suivante: ***(Checksum = Encrypt((checksum+SignP(X) +P), clé pub. plate-forme d'origine))).*** Si la confidentialité est nécessaire, l'objet est crypté avec la clé publique de la plate-forme d'origine avant son insertion dans le vecteur. Quand l'agent retourne chez lui, la plate-forme d'origine peut vérifier si les éléments du journal ont été frauduleusement modifiés. Le processus fait le travail à l'inverse, décrypte la somme de contrôle et vérifie la signature de chaque élément dans le journal. Si une erreur est trouvée, la plate-forme d'origine saura que l'objet en question a été frauduleusement modifié.

Révélation sélective

Dans certaines applications, l'agent mobile peut décider de ne rendre accessible une partie de ces données qu'à l'intérieur d'un ensemble de plates-formes préalablement fixé. Pour cela, l'agent mobile contient un vecteur d'objets qui représentent cette partie de données. Chacun de ces objets est crypté par la clé publique de la plate-forme à laquelle l'objet est destiné. Une fonction de hashage est appliquée à ce vecteur. Le résultat est ensuite signé par la plate-forme d'origine de l'agent mobile. Ce vecteur d'objets est généré par celle-ci avant que l'agent mobile n'entreprenne son voyage.

Quand l'agent arrive dans une plate-forme, celle-ci cherche dans ce vecteur si elle a des objets qui lui sont destinés. Si c'est le cas, elle se charge de les décrypter avec sa clé privée. La plate-forme doit vérifier la signature de l'agent avant de décrypter les objets. Si une plate-forme malveillante change le contenu de ce vecteur, la signature sera invalide.

2.2.4 Protection d'agents mobiles en utilisant les états de référence

Hohl [HOH00] définit une attaque comme une différence dans le comportement entre la plate-forme attaquante et une plate-forme non-attaquante ou de référence i.e. une plate-forme qui agit comme prévu (*comportement de référence*). Il dénote avec le mot «*comportement*» la façon dont le système doit se comporter afin d'exécuter un agent. Si ce comportement diffère des spécifications de l'agent mobile, le système agit d'une manière non prévue par le programmeur. La définition d'une attaque ci-dessus mène à une protection où la différence dans le comportement est mesurée pour prouver ou détecter le comportement malveillant. Il y a deux problèmes qui limitent la praticabilité de cette solution. D'une part, une partie du comportement de la plate-forme ne peut pas être observée de l'extérieur. Pratiquement, il est trop difficile de contrôler toutes les actions futures d'une plate-forme. D'autre part, il est difficile de fournir à la plate-forme de référence l'état et les entrées d'une plate-forme malveillante.

Un état de référence comprend les parties variables d'un agent mobile exécuté par une plate-forme de référence. Si on peut mesurer la différence des états de l'agent mobile, on peut détecter les attaques qui génèrent un état différent de l'état de référence. Ces attaques incluent celles où la plate-forme attaquante modifie les parties variables de l'agent mobile et celles où le code de l'agent est exécuté d'une manière incorrecte. Les attaques qui ne peuvent pas être détectées par cette approche sont les attaques d'espionnage du code et des données.

Trois principales approches utilisent les états de référence pour détecter les attaques : l'*évaluation d'état*, les *serveurs répliqués* et *la trace d'exécution*. La première approche (évaluation d'état) est décrite dans [FAR96b] par Fermier *et al.* Il permet à la

plate-forme qui reçoit l'agent mobile de contrôler la validité de son état avant son exécution. Ce mécanisme de contrôle considère seulement l'état actuel de l'agent. Le mécanisme peut consister par exemple en un ensemble de conditions qui doivent être satisfaites après la session d'exécution. Dans ce cas-ci, les données de référence sont structurées en un ensemble de règles. Celles-ci sont formulées par le programmeur.

La deuxième approche, appelée serveurs répliqués, est décrite dans [MIN96]. Ils proposent d'employer un mécanisme de tolérance aux pannes pour détecter les attaques des plates-formes malveillantes. Les plates-formes répliquées sont des plate-formes qui offrent le même ensemble de ressources i.e. les mêmes données, mais ne partagent pas le même intérêt en attaquant un agent mobile i.e. elles appartiennent à des entreprises différentes. Chaque étape d'exécution est traitée en parallèle par toutes les plates-formes répliquées. Après l'exécution, les plates-formes votent au sujet du résultat de l'étape courante. L'exécution qui a obtenu le plus grand nombre de voix gagne, et la prochaine étape est exécutée. Puisque les plates-formes fonctionnent en parallèle, les entrées de l'agent mobile doivent être partagées. L'approche de serveur répliqué peut détecter toutes les attaques qui génèrent un état différent de l'état de référence.

La troisième approche, dite trace d'exécution, est présentée par Vigna [VIG98]. Il propose une technique qui permet de détecter toute exécution anormale d'un agent sur une plate-forme à partir d'un fichier, une sorte de « résumé de l'exécution » appelé *trace*. Ces traces sont signées par chaque plate-forme, ce qui permet de retrouver la plate-forme coupable d'une mauvaise exécution de l'agent. À la fin de l'exécution de celui-ci, si l'utilisateur de l'agent a des doutes quant à sa bonne exécution, il « réexécute » l'agent grâce aux traces et, en cas d'erreur, il peut ainsi retracer la plate-forme coupable. Une trace est une paire (n,s) où n désigne un identifiant unique d'une ligne de code et s la signature de cette ligne de code. Pour les opérations qui ne modifient les variables que par rapport à d'autres variables internes (opérations blanches), la signature est vide. Le Tableau 2.2 illustre la notion de trace.

Tableau 2.2 Exemple de trace

Code	Trace
10. read(x)	(10, x = 5)
11. y = x+z	(11,-)
12. m = y+1	(12,-)
13. k = cryptInput	(13, k = 2)
14. m = m+k	(14,-)

Après l'exécution, une plate-forme crée un fichier en appliquant une fonction de hashage aux traces de l'exécution. Ce fichier est ensuite signé par la plate-forme grâce à sa clé privée. Puis, elle envoie ce fichier à la prochaine plate-forme de l'agent avec le code et l'état de l'agent. Une fois que l'agent a terminé son exécution et est rentré sur sa plate-forme d'origine, cette dernière peut décider si elle veut vérifier l'exécution de l'agent. Si c'est le cas, elle ré-exécute l'agent à partir de son état initial, puis compare le résultat de la fonction de hashage de ses traces avec celui de l'exécution à distance. S'ils sont égaux, c'est que la plate-forme n'a pas triché et on passe à la plate-forme suivante. Cette approche détecte toutes les attaques qui donnent un état différent, du moment que la plate-forme ne ment pas sur les entrées.

Le principal élément qui entre en jeu pour faire l'analyse de ces approches est le moment où on effectue le contrôle. La différence dans les états peut être contrôlée de deux façons : a) après chaque session d'exécution dans une plate-forme ; b) ou bien après que l'agent a terminé sa tâche. Ce dernier choix signifie que l'itinéraire de l'agent doit être enregistré d'une manière sécuritaire. Les données de référence utilisées doivent être enregistrées pour chaque session d'exécution. Sans cette précaution, la plate-forme malveillante ne peut pas être identifiée.

Les mécanismes utilisant les états de référence ne peuvent pas détecter toutes les attaques possibles par les plates-formes malveillantes. Les attaques qui n'ont pas comme résultat un état d'agent différent ne peuvent pas être détectées par ces approches.

Particulièrement les attaques d'espionnage de données ou du code ne sont pas détectables.

2.2.5 Boîte noire sécuritaire limitée dans le temps

L'idée principale de cette approche [HOH98] est de générer un agent exécutable à partir des spécifications d'un agent mobile initial qui ne peut être menacé par ni les attaques de lecture ni celles de la manipulation. On suppose que la protection n'est pas permanente mais a une durée limité connue d'avance. L'agent est considéré comme une « *Boîte noire limitée dans le temps*» s'il satisfait les conditions suivantes :

- un agent est une boîte noire si :
  a) pour un certain intervalle de temps connu,
  b) le code et les données de l'agent initial ne peuvent pas être lus,
  c) le code et les données de l'agent initial ne peuvent pas être modifiés,
- les attaques après l'intervalle de protection sont possibles mais elles n'ont aucun effet.

Si cette définition peut être appliquée à un agent, seules les entrées et les sorties de la boîte noire sont observables. L'approche utilisée dans la boîte noire est basé sur les fonctions cryptographiques décrites par Sander et Tschudin [SAN98]. Le mécanisme de conversion qui génère l'agent avec la boîte noire utilise des paramètres de configuration qui permettent de créer différentes sorties à partir d'une même spécification. Ces paramètres prévoient l'attaque avec la force brute. La Figure 2.5 illustre l'idée de base de cette approche.

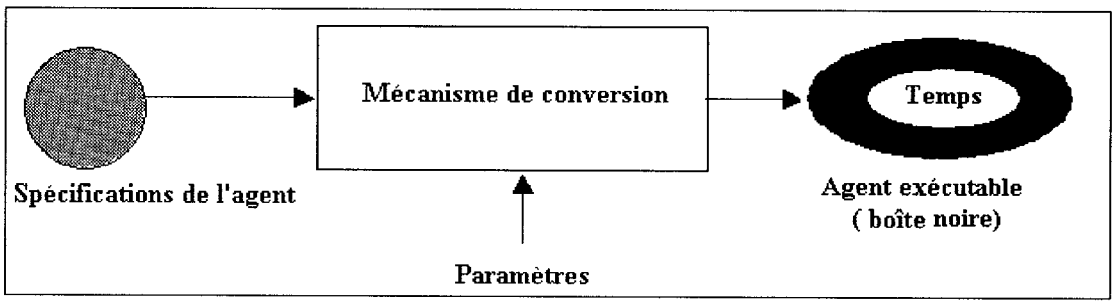

Figure 2.5 Boîte noire limitée dans le temps

Le principal problème que rencontrent les approches qui protègent l'agent mobile contre la plate-forme est basé sur le fait que celle-ci est toujours capable de lire chaque bit de sa mémoire et le contenu de chaque variable, et de savoir le contenu de chaque ligne du code. Mais, si on veut comprendre le sens des instructions et les relations entre les variables d'un code, on doit réfléchir sur la sémantique du code. Cette sémantique n'est pas exprimée par le code, mais elle peut être devinée par un « modèle mental » d'un programmeur ou d'un lecteur du programme.

L'idée principale de cette approche est d'interdire à un attaquant de bâtir un tel modèle mental de l'agent mobile avant l'arrivée de celui ci, et de rendre ce processus une tâche coûteuse en terme de temps d'exécution. Le premier objectif est atteint en créant une nouvelle forme de l'agent dynamiquement. Le deuxième est atteint en utilisant des algorithmes de conversion qui produisent des formes d'agent difficiles à analyser. On les appelle des *algorithmes de confusion* (*mess-up algorithm*). Le rôle de ce type d'algorithme est de générer un nouvel agent à partir d'un agent initial qui a un code et des données différents mais aboutit au même résultat. Pour se protéger contre l'attaque à force brute, l'algorithme utilise un paramètre aléatoire en entrée. Le concepteur de ce genre d'algorithme doit prendre en considération deux aspects : les attributs de l'agents qui peuvent être modifiés et les caractéristiques de l'attaquant.

Les attributs modifiables de l'agent sont les instructions et les données. Une instruction a un type et une localisation dans un programme. Le type peut être caché jusqu'à ce que l'instruction soit exécutée dynamiquement. La localisation peut être aussi cachée, soit implicitement en utilisant la création du code dynamique, soit explicitement en camouflant certaines instructions dans d'autres.

Une donnée (variable ou constante) consiste en le type, la valeur et la localisation. Le type peut être caché jusqu'à ce qu'on ait besoin de la donnée. La valeur d'un élément de la donnée peut être cachée. Une façon de faire est de remplacer les accès à cet élément par des accès à des sous-élément et de transformer les opérations sur cet élément à des opérations sur des sous-éléments. Finalement, la localisation d'une

donnée peut être cachée statiquement en divisant l'élément et en distribuant ses parties. Quant au deuxième aspect, la modélisation de l'attaquant, on distingue deux cas. Dans le premier cas, l'attaquant ne sait rien sur la version originale de l'agent initial. Un humain doit analyser la boîte noire afin de bâtir un modèle mental. Dans le deuxième cas, l'attaquant a des connaissance sur la spécification exacte de l'agent en avance. Il peut automatiser ces attaques en générant un programme qui essaye de calculer certains attributs de l'agent.

Comme exemple d'algorithme de confusion, on cite la *recomposition de variables*. Cet algorithme prend l'ensemble des variables de programme, divise chaque variable en des segments et crée des nouvelles variables qui contiennent une recomposition des segments originaux. Le code d'accès à ces nouvelles variables peut être créé automatiquement en utilisant des fonctions de conversion. La Figure 2.6 illustre l'algorithme de recomposition de variables.

Le principal problème des algorithmes de confusion est que l'intervalle de temps de protection doit être d'une longueur utilisable. S'il est très long, l'agent peut migrer vers beaucoup de site et faire des calculs nécessitant un très grand temps d'exécution. Si cet intervalle est très petit, le domaine des applications de l'agent protégé est sévèrement restreint.

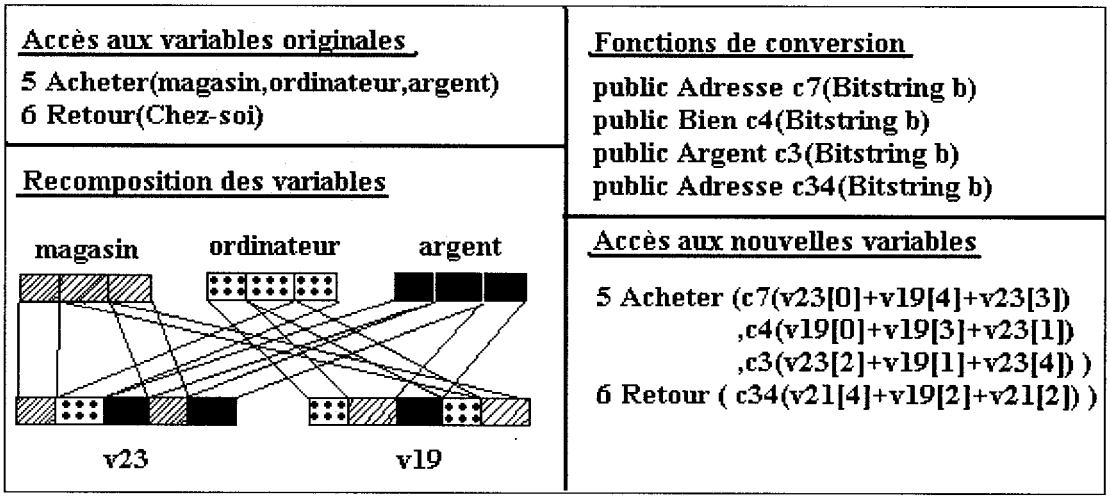

Figure 2.6 Recomposition de variables

Le coût de la sécurité avec la boîte noire limitée dans le temps est composé de coût de la création, de coût de la transmission et de coût de l'exécution. Le coût de la création est causé par la conversion de l'agent original vers sa nouvelle forme. Le coût de la transmission vient du fait que l'agent doit transporter toutes ses librairies de la sécurité au lieu d'utiliser celles du système de la plate-forme distante. Le dernier coût, celui de l'exécution, résulte d'une part des calculs introduits par les algorithmes de confusion; et d'autre part, du temps d'exécution relatif à la librairie de sécurité transportée.

2.2.6 Protection mutuelle des agents coopérants

L'agent mobile effectue des déplacements d'une plate-forme à une autre afin d'accomplir la tâche qu'on lui demande d'exécuter. Ces déplacements définissent l'itinéraire de l'agent. Lors de son voyage, un agent peut visiter une plate-forme malveillante qui peut modifier son itinéraire en lui imposant un certain nombre de sites à visiter ou en lui interdisant la visite de certain d'autres. Le but du protocole de Roth [ROT98] est la détection de ce type d'attaques. Ce protocole est basé sur la coopération d'un autre agent qui se lance au même moment que l'agent original et qui a pour rôle l'enregistrement de l'itinéraire de l'agent et la vérification de ce dernier au fur et à mesure que l'agent avance dans son chemin.

L'idée de Roth est basée sur l'hypothèse que dans un environnement où l'agent se déplace, il existe un ensemble de plates-formes malveillantes qui ne sont pas près à collaborer avec d'autres plates-formes pour attaquer l'agent mobile. Le modèle devise l'ensemble des plates-formes en fonction de leurs habilités à collaborer dans une attaque contre l'agent mobile. Soit H un ensemble de plates-formes interconnectées via un réseau. Pour une instance d'un agent, R dénote un sous-ensemble de $H \times H$ tel que $(h_i, h_j) \in R \Leftrightarrow h_i$ et h_j collaborent dans le but d'attaquer l'agent. Soit H_α et H_β deux sous-ensembles non vides de H avec $(H_\alpha \times H_\beta) \cap R = \phi$. Les sous-ensembles H_α et H_β sont dits non collaborant. On dit que les deux agents α et β sont coopérants lorsque l'itinéraire de

α ne contient que des plates-formes de H_α et que l'itinéraire de l'agent β ne contient que des plates-formes de H_β.

Pour que le protocole de la protection mutuelle entre agents coopérants soit applicable, les hypothèses suivantes doivent être faites :

- Le déplacement de l'agent entre deux plate-formes se fait via un canal authentifié;
- Les plates-formes fournissent un canal de communication authentifié aux agents coopérants;
- L'agent a un accès, de manière authentifiée, aux identités des plates-formes dont il a besoin.

Le protocole d'enregistrement d'itinéraire se résume de la manière suivante. Soit α et β deux agents coopérants, H_α et H_β deux ensembles de plates-formes qui sont inclus dans H et non collaborant. L'agent β enregistre et vérifie l'itinéraire de l'agent α. Soit h_i $\in$ H_α, la $i^{ème}$ plate-forme visitée par l'agent α et soit $id(h_i)$ son identité. Soit $Pred_i$, l'identité de la dernière plate-forme où l'agent mobile s'est exécuté. Enfin, soit $Suiv_i$ l'identité de la prochaine plate-forme où l'agent désire aller après la plate-forme h_i. L'agent débute sur la plate-forme h_0. Ainsi, si l'agent se déplace n fois, on a $h_0 = h_n$ puisque l'agent revient sur la plate-forme d'origine.

Initialisation :

Soit h_0 la plate-forme d'origine des agents α et β. h_0 doit être une plate-forme de confiance pour α et β. Pour les deux agents, $Suiv_i$ a pour valeur l'identité de la plate-forme sur laquelle les agents vont s'exécuter. Puis, chacun des agents est envoyé vers leur première plate-forme par l'intermédiaire d'un canal de communication authentifié.

Étape i, i {1,…,n} :

L'agent α envoie un message à β contenant $Suiv_i$ et $Pred_i$ à travers un canal authentifié. Ainsi, l'agent β apprend $id(h_i)$. L'agent β vérifie que $\boldsymbol{Id(h_i) = Suiv_{i-1}}$ ***et*** $\boldsymbol{Pred_i = id(h_{i-1})}$. Si c'est le cas, il enregistre $Suiv_i$ dans l'itinéraire de l'agent.

L'attaque que le protocole détecte correspond à celle où la plate-forme h_i envoie l'agent α vers une plate-forme h_{i+1} telle que $id(h_{i+1}) \neq Suiv_i$. La plate-forme h_i a deux

manières d'effectuer cette attaque. Elle peut soit modifier la communication entre l'agent α et l'agent β (cette attaque est détectée au retour de l'agent α sur sa plate-forme d'origine), soit ne pas modifier la communication et envoyer l'agent α vers une plate-forme autre que celle où l'agent veut aller (cette attaque est immédiatement détectée).

Allée [ALL01] a proposé une amélioration du protocole de Roth. Il propose de déplacer l'agent β à chaque fois que l'agent α se déplace i.e. lorsque α contactera β pour lui annoncer l'identité de la prochaine plate-forme où il va s'exécuter et celle de la plate-forme précédente qu'il a visitée. Une fois ces informations reçues, l'agent β va se déplacer vers une nouvelle plate-forme et attendre que l'agent α communique avec lui. Il suppose que l'agent α possède une liste ordonnée des plates-formes sur lesquelles l'agent β va aller. Il suppose aussi que l'ensemble des plates-formes visitées par l'agent α et celui des plates-formes visitées par l'agent β sont *disjoints*.

2.3 Comparaison des approches

Le Tableau 2.3 suivant présente les protections possibles contre les différentes attaques énumérées au début de ce chapitre que les approches actuelles ont pu assurer. Les approches étudiées peuvent être classées en deux catégories ; celles qui détectent ou protègent l'agent mobile contre les attaques visant son code, et celles qui détectent ou protègent l'agent mobile contre les attaques visant ses données. Seule la cinquième approche (Boîte noire sécuritaire limitée dans le temps) a pu combiner les deux types de protection, mais son principal défaut est la façon dont elle a utilisé le facteur temps qui implique la synchronisation de tous les serveurs avec un temps global pour que le protocole puisse fonctionner correctement.

Tableau 2.3 Comparaison des approches

	Protection crypto.	**Sécurité crypto.**	**Sécurité de données**	**États de référence**	**Boîte noire limitée**	**Agents coopérant**
Espionnage du code	Protection	Protection			Protection	
Espionnage de données			Protection		Protection	
Espionnage de l'état	Protection	Protection				
Manipulation du code	Protection	Protection		Détection	Protection	
Manipulation des données			Détection		Protection	
Manipulation de l'état	Protection	Protection	Détection	Détection		Détection
Exécution incorrecte du code	Protection	Protection		Détection	Protection	
Mascarade de la plate-forme			Protection			
Déni de service					Détection	
Espionnage des interactions avec les autres agents					Protection	
Manipulation des interactions avec les autres agents				Détection		
Retour des résultats erronés des appels systèmes effectués par l'agent				Détection	Protection	
Ré-exécution de l'agent						

Après avoir analysé ces différentes approches, nous constatons que l'attaque qui consiste à ré-exécuter l'agent n'est traitée par aucune approche. Les attaques déni de service et manipulation des interactions avec les autres agents n'ont fait l'objet que des travaux de détection. Et les approches, généralement, ne protégent qu'une partie de l'agent (soit son code, soit ses données). D'où l'importance de concevoir un protocole plus sécuritaire que ceux existant qui protège l'agent mobile contre toutes ces attaques.

CHAPITRE III
SPECIFICATIONS ET SECURITE DU PROTOCOLE

L'approche que nous préconisons, dans la conception d'un protocole sécuritaire protégeant l'agent mobile contre les attaques des plates-formes malveillantes, est basée sur la coopération complète d'un agent sédentaire s'exécutant à l'intérieur d'une tierce plate-forme fiable. Ce protocole a pour objectif de protéger une partie du code de l'agent mobile contre la modification et les exécutions incorrectes, de détecter les attaques de modification des résultats intermédiaires obtenus à l'intérieur des plates-formes visitées, d'interdire la ré-exécution du code de l'agent mobile, de protéger celui-ci contre l'attaque appelée *déni de service* et d'enregistrer l'itinéraire de l'agent mobile afin de détecter les attaques liées à sa modification. Dans ce chapitre, nous allons tout d'abord décrire les spécifications de notre protocole sécuritaire en précisant les échanges de messages entre l'agent mobile et son agent sédentaire. Ensuite, nous détaillerons les différents scénarios d'attaques et nous expliquerons la manière selon laquelle notre protocole permet la protection de l'agent mobile contre ces attaques.

3.1 Définitions et hypothèses

Le concept de base de notre protocole est fondé sur la coopération d'un agent sédentaire pour protéger l'agent mobile. Celui-ci effectue son parcourt à travers plusieurs plates-formes, et à chaque étape d'exécution à l'intérieur d'une plate-forme visitée, il envoie un certain nombre de messages à l'agent coopérant qui se charge de détecter les attaques contre le code et les données de l'agent mobile.

3.1.1 Intervenants dans notre protocole

Notre protocole fait intervenir cinq éléments principaux : la plate-forme d'origine (O), l'ensemble (P) des plates-formes visitées par l'agent mobile, la tierce plate-forme fiable (T), l'agent mobile (AM) et l'agent sédentaire appelé aussi l'agent

coopérant (*AS*). La plate-forme d'origine *O* constitue le point de départ de tout le mécanisme. Elle se charge de générer l'agent mobile en fonction des demandes formulées par l'utilisateur. Elle se charge aussi de générer l'agent sédentaire (*AS*). L'ensemble (*P*) est constitué de *L* plates-formes P_i (i=1, …, L) où l'agent mobile exécute son code. Chaque plate-forme P_i possède un identifiant unique (Id_i) qui l'identifie sans ambiguïté. Dans la $i^{ème}$ exécution de l'agent mobile (*AM*), $Prec_i$ désigne l'identité de la plate-forme où l'agent mobile provient et $Suiv_i$ désigne l'identité de la plate-forme où l'agent mobile veut migrer. La Figure 3.1 illustre ces différents intervenants.

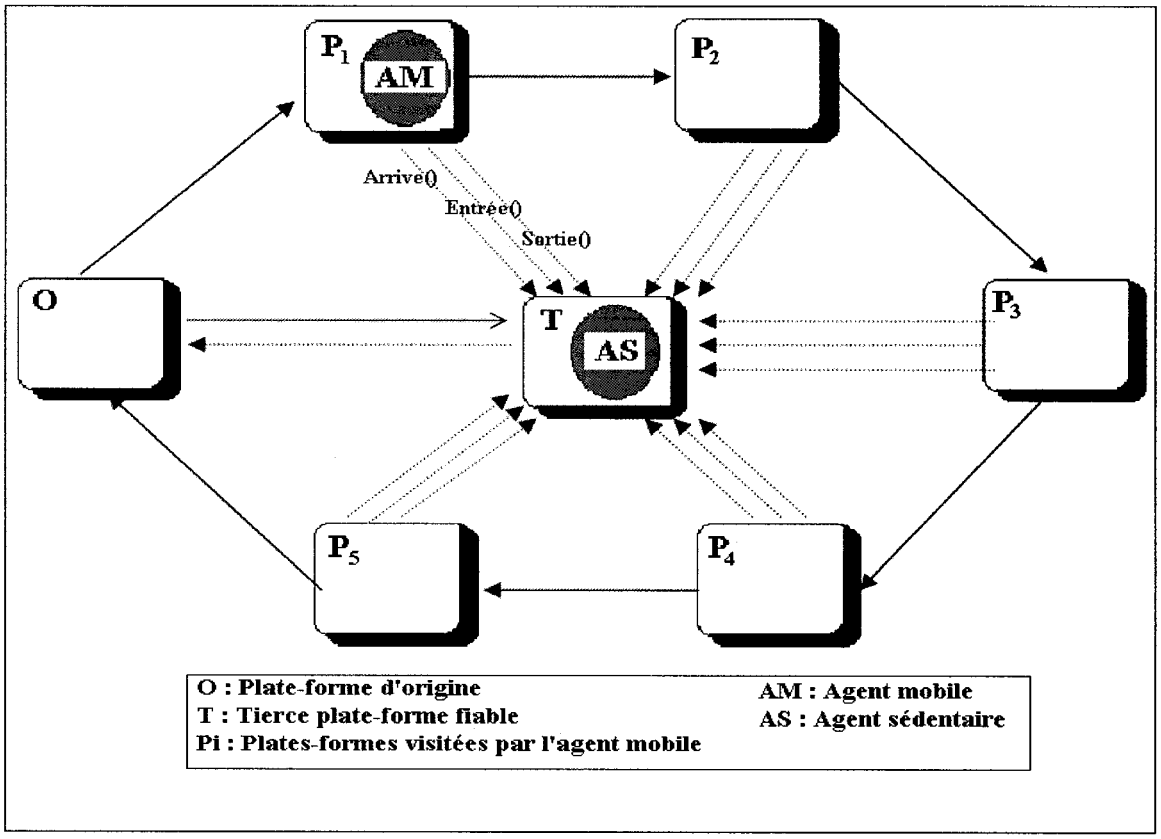

Figure 3.1 Intervenants du protocole

Nous notons par $E_S(X)$ l'opération d'encryptage d'une donnée *X* avec la clé publique de la plate-forme *S*. L'opération de décryptage d'une donnée *X* par la clé privée

de la plate-forme S est notée par $D_S(X)$. $SIG_S(X)$ désigne la signature électronique d'une donnée X par la plate-forme S.

3.1.2 Hypothèses émises

Pour que notre protocole puisse fonctionner correctement, nous faisons trois hypothèses : la disponibilité d'une tierce plate-forme fiable, l'établissement des canaux de communication entre l'agent mobile et l'agent sédentaire, et l'existence d'un système cryptographique asymétrique. Dans la première hypothèse, nous supposons qu'une tierce plate-forme fiable (T) est mise à la disposition de toutes les autres plates-formes pour exécuter l'agent coopérant (AS). Cette plate-forme doit effectuer des exécutions correctes du code. Elle ne doit collaborer ni avec la plate-forme d'origine (O) contre les autres plates-formes (P_i), ni avec une plate-forme (P_i) contre la plate-forme d'origine (O) ou contre une plate-forme P_j ($i \neq j$). Dans le domaine des applications commerciales, ce service peut être payable. La tierce plate-forme fiable (T) sera obligée de respecter les conditions mentionnées dans cette hypothèse, sinon elle perdra son image de marque et ses clients.

La validité de cette hypothèse est justifiée par l'existence réelle des applications réparties déployées à travers le réseau Internet et qui utilisent des serveurs fiables pour leur bon fonctionnement. Nous citons à titre d'exemple l'infrastructure PGP [ZIM95] qui fournit des services cryptographiques à l'aide d'un système asymétrique (clé privée/clé publique). La deuxième hypothèse stipule qu'au début de son exécution dans une plate-forme P_i, l'agent mobile établit un canal de communication fiable avec l'agent sédentaire. Si la plate-forme P_i refuse la création de ce canal, l'agent mobile réalise que cette plate-forme est une plate-forme attaquante. Dans la dernière hypothèse, nous supposons que les plates-formes O, P_i (i=1, ..., L) et T utilisent un système cryptographique asymétrique (clé privée/clé publique). Toutes les plates-formes P_i (i=1, ..., L) autorisent l'agent mobile à effectuer l'encryptage des données transmises.

3.2 Le protocole

Le code de l'agent mobile est subdivisé en deux parties. Une partie critique, appelée *code critique*, contient le code de l'agent mobile dont la manipulation influence l'intégrité de l'agent mobile. C'est la partie du code qui implémente, à titre d'exemple, la comparaison des offres des plates-formes visitées par l'agent mobile. La deuxième partie, dite non critique, contient le reste du code de l'agent mobile. Une plate-forme malveillante n'aura aucun intérêt à attaquer cette partie du code et, même si elle l'attaque, elle ne perturbera pas le bon fonctionnement de l'agent mobile.

La sécurité du code critique est basée sur sa duplication dans les deux agents (mobile et sédentaire) qui exécutent simultanément cette partie du code et comparent leurs résultats. Le protocole empêche une plate-forme P_i de ré-exécuter l'agent mobile et détecte l'attaque déni de service en utilisant deux compteurs de temps : un délai de garde (*Timeout*) et un intervalle d'attente supplémentaire (*IAS*). Il permet aussi de préserver l'itinéraire de l'agent mobile et de détecter toute attaque visant à le modifier en s'inspirant du protocole de Roth [ROT98].

3.2.1 Initialisation du protocole

Au début du protocole, la plate-forme d'origine (O) génère l'agent mobile et son agent coopérant. La partie sensible du code de l'agent mobile est marquée comme étant sa partie critique. Cette partie est dupliquée dans l'agent sédentaire. Puis, la plate-forme d'origine initialise les variables des deux agents; plus particulièrement, elle affecte à $Prec_0$ la valeur de son identité. Elle envoie l'agent mobile à la première plate-forme (P_1) de son itinéraire et elle envoie l'agent coopérant (AS) à la tierce plate-forme (T) où il demeure stationnaire jusqu'à la fin de son exécution. La Figure 3.2 illustre les étapes d'initialisation du protocole.

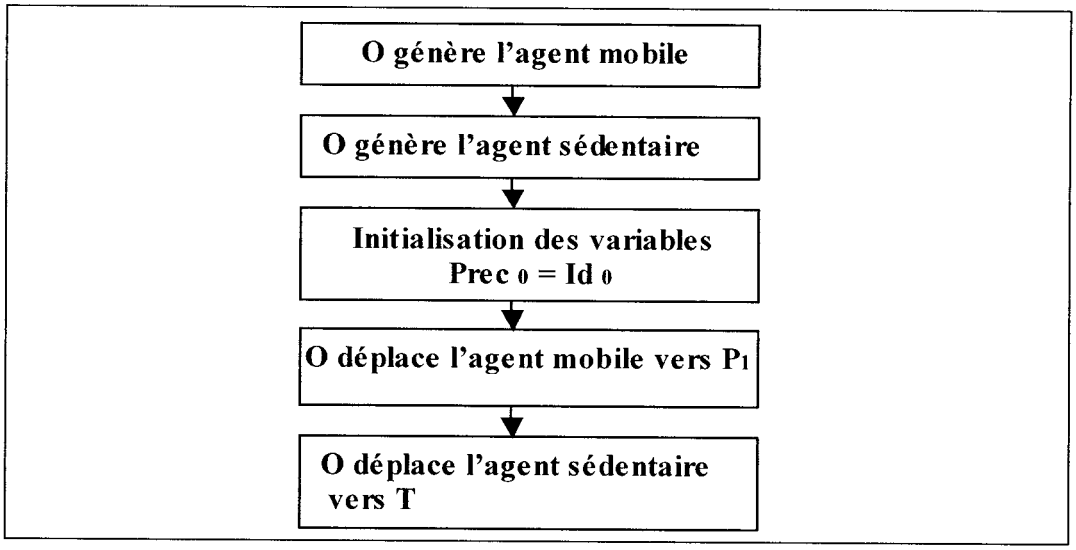

Figure 3.2 Initialisation du protocole

3.2.2 Étape i (i=1, ..., L)

Quand l'agent mobile arrive à la $i^{ème}$ plate-forme P_i, il envoie un message de type *Arrive()* à son agent coopérant (*AS*) afin de lui signaler son arrivée à la plate-forme P_i. Le message *Arrive()* contient l'identité (Id_i) de la plate-forme P_i, l'identité de l'agent mobile (Id_{AM}) signée par la plate-forme P_i et l'identité ($Prec_i$) de la plate-forme d'où provient l'agent mobile. La Figure 3.3 illustre la structure du message *Arrive()*.

Arrive () :
- **Id_i** : Identité de la plate-forme P_i qui a transmis le message
- **$SIG_{Pi}(Id_{AM})$** : La signature de la plate-forme P_i sur l'identité de l'agent mobile
- **$Prec_i$** : L'identité de la plate-forme P_{i-1} d'où provient l'agent mobile

Figure 3.3 Structure du message *Arrive()*

Quand l'agent sédentaire reçoit le message *Arrive()*, il vérifie la signature

électronique de la plate-forme P_i sur l'identité de l'agent mobile pour s'assurer que le message provient bien de la plate-forme P_i. Puis, il vérifie l'itinéraire de l'agent mobile en comparant l'identité de la plate-forme courante P_i avec la valeur contenue dans $Suiv_{i-1}$ et l'identité de la plate-forme précédente P_{i-1} avec la valeur contenue dans $Prec_i$ afin de s'assurer que la plate-forme P_i est bien celle où l'agent mobile veut migrer. Les égalités vérifiées par l'agent sédentaire sont:

$$\boldsymbol{Id_i = Suiv_{i-1}}$$

et

$$\boldsymbol{Prec_i = Id_{i-1}.}$$

Si les égalités précédentes sont vérifiées, l'agent sédentaire réalise que l'agent mobile *AM* a atterri au bon endroit. Dans ce cas, l'agent sédentaire initialise un compteur de temps *Timeout* qui servira à limiter le temps d'exécution de l'agent mobile à l'intérieur de la plate-forme P_i. Celle-ci commence l'exécution du code de l'agent mobile.

Entrée() :

- **Id_i** : l'identité de la plate-forme P_i
- **X** : les données fournies par P_i, requises pour exécuter le code critique
- **$SIG_{Pi}(X)$** : la signature sur les données X par la clé publique de la plate-forme P_i

Figure 3.4 Structure du message *Entrée*()

Avant d'exécuter la partie du code critique, l'agent mobile communique avec l'agent sédentaire et lui envoie un message de type *Entrée*() contenant toutes les entrées requises fournies par la plate-forme courante P_i. Le message *Entrée()* est composé de l'identité (Id_i) de la plate-forme courante P_i, des données (X) requises pour l'exécution de la partie du code critique et de la signature $SIG_{Pi}(X)$ sur les données (X) par la clé privée de la plate-forme P_i. Avant sa transmission, le message *Entrée()* est crypté par la

clé publique de la tierce plate-forme fiable (T). La plate-forme P_i se charge d'envoyer $E_T(Entrée(Id_i, SIG_{Pi}(X),X))$ à l'agent sédentaire AS. La Figure 3.4 illustre les composants du message *Entrée()*.

Après l'envoi du message *Entrée()*, l'agent mobile continue l'exécution de son code. Lorsque l'agent sédentaire reçoit le message $E_T(Entrée())$, il décrypte le contenu de celui-ci avec sa clé privée en calculant $D_T(E_T(Entrée()))$. Ensuite, il vérifie l'exactitude de l'identité de la plate-forme P_i en vérifiant la signature $SIG_{Pi}(X)$ sur les données (X). Puis, il commence l'exécution de la partie du code dupliquée de l'agent mobile. Quand l'agent mobile termine l'exécution de son code, il envoie un message de type *Sortie*() à l'agent sédentaire. Le message *Sortie()* est composé de l'identité (Id_i) de la plate-forme courante P_i, des résultats (R) obtenus par l'agent mobile à l'intérieur de la plate-forme P_i, de la signature $SIG_{Pi}(R)$ sur ces résultats avec la clé privée de la plate-forme P_i et de l'identité $Suiv_i$ de la plate-forme P_{i+1} où l'agent mobile veut migrer. La Figure 3.5 illustre les éléments constituant le message *Sortie()*.

Sortie() :

- **Id_i** : l'identité de la plate-forme P_i
- **R** : les résultats obtenus par l'agent mobile suite à l'exécution de sa partie du code critique
- **$SIG_{Pi}(R)$** : la signature sur les données R par la clé publique de la plate-forme P_i
- **$Suiv_i$** : l'identité de la plate-forme P_{i+1} où l'agent mobile veut migrer

Figure 3.5 Structure du message *Sortie()*

L'agent mobile encrypte le message *Sortie()* avec la clé publique de la tierce plate-forme (T) et transmet $E_T(Sortie(Id_i, SIG_{Pi}(R), R, Suiv_i))$ à l'agent sédentaire. Quand celui-ci reçoit ce message, il le décrypte avec sa clé privée en calculant $D_T(E_T(Sortie()))$ et obtient le message *Sortie()*. Puis, il vérifie l'exactitude de l'identité de la plate-forme émettrice du message en vérifiant sa signature $SIG_{Pi}(R)$ sur les résultats. Ensuite, l'agent sédentaire procède à la vérification de l'exécution du code critique de l'agent mobile en

comparant les résultats obtenus par celui-ci à l'intérieur de la plate-forme P_i avec les résultats obtenus par l'agent sédentaire qui sont corrects vu que la tierce plate-forme (T) est supposée exécuter le code d'une manière exacte. Si les résultats obtenus par l'agent coopérant sont différents de ceux obtenus par l'agent mobile, l'agent sédentaire conclut que la plate-forme courante P_i a attaqué l'agent mobile. L'agent sédentaire marque cette plate-forme comme étant une plate-forme attaquante. Si les deux résultats sont concordants, l'agent coopérant conclut que l'exécution de la partie du code critique s'est déroulée sans aucune attaque. Dans les deux cas, l'agent mobile continue son itinéraire vers la plate-forme suivante. L'agent sédentaire enregistre l'itinéraire de l'agent mobile en sauvegardant l'identité $Suiv_i$ de la plate-forme P_{i+1} où l'agent mobile veut migrer. La Figure 3.6 illustre les trois types de messages échangés entre l'agent mobile et l'agent sédentaire.

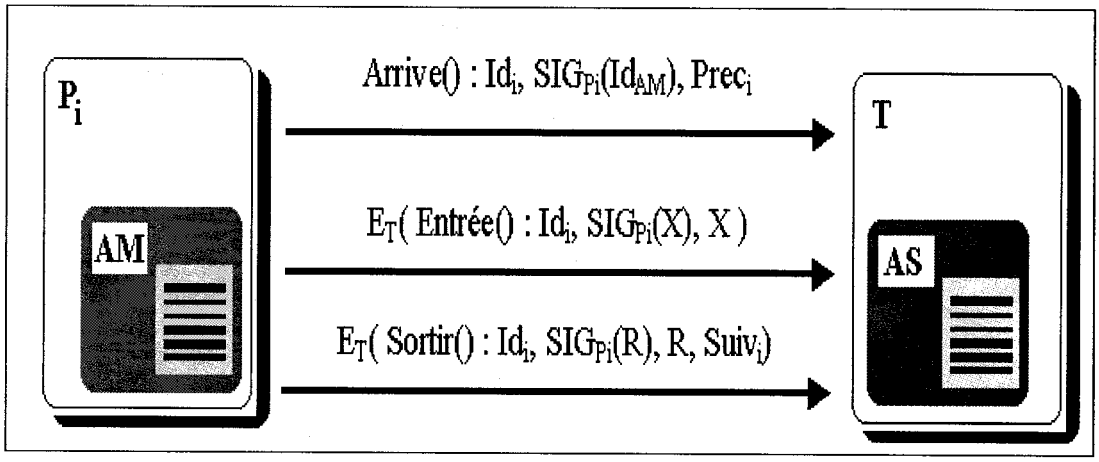

Figure 3.6 Messages échangés entre l'agent mobile et l'agent sédentaire

Dés la réception du message *Arrive()*, l'agent sédentaire initialise un *Timeout* à une valeur appropriée présentant le temps maximum nécessaire à l'exécution de la totalité du code de l'agent mobile et au transfert des deux messages *Entrée()* et *Sortie()* entre l'agent mobile et l'agent sédentaire. Si la valeur de *Timeout* arrive à son expiration (*Timeout* = 0) avant que l'agent sédentaire n'ait reçu le message *Sortie()* de la part de son agent mobile, l'agent sédentaire déduit que la plate-forme P_i est une plate-forme

attaquante, les résultats obtenus par l'agent mobile à l'intérieur de cette plate-forme seront ignorés par la plate-forme d'origine *O*. L'agent sédentaire continue à attendre l'arrivée du message *Sortie()* pendant un autre intervalle de temps supplémentaire, même après l'expiration de *Timeout*. Après l'écoulement de cet intervalle de temps d'attente, l'agent sédentaire conclut que l'attaque déni de service a eu lieu, il avertit la plate-forme d'origine *O* pour qu'elle puisse relancer le protocole en omettant la plate-forme P_i de l'itinéraire de l'agent mobile. Ensuite, il donne l'ordre à l'agent mobile d'arrêter son exécution si celle-ci est encore en cours. Puis, l'agent sédentaire termine son exécution à l'intérieur de la tierce plate-forme *T*. Si le message *Sortie()* arrive dans l'intervalle du temps d'attente, l'agent sédentaire marque la plate-forme P_i comme attaquante et laisse l'agent mobile continuer son itinéraire. Les figures 3.7 et 3.8 illustrent l'algorithme suivi par l'agent mobile et celui suivi par l'agent sédentaire respectivement.

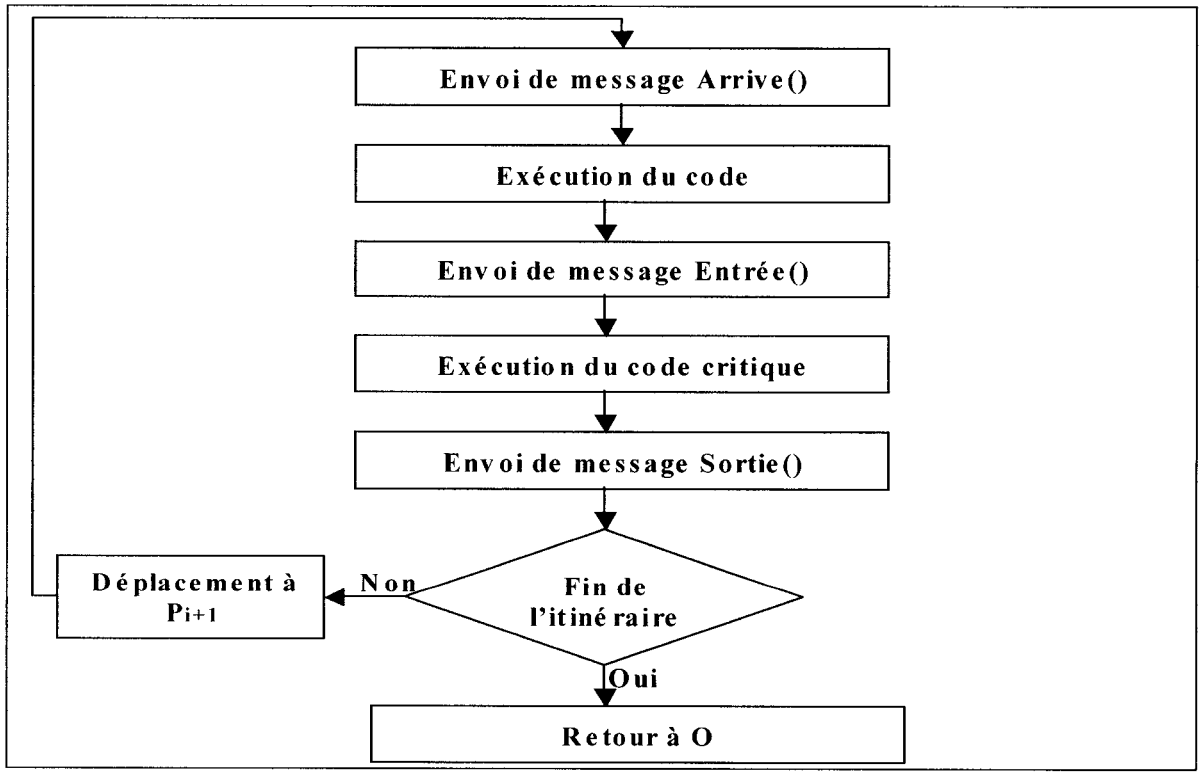

Figure 3.7 Algorithme suivi par l'agent mobile

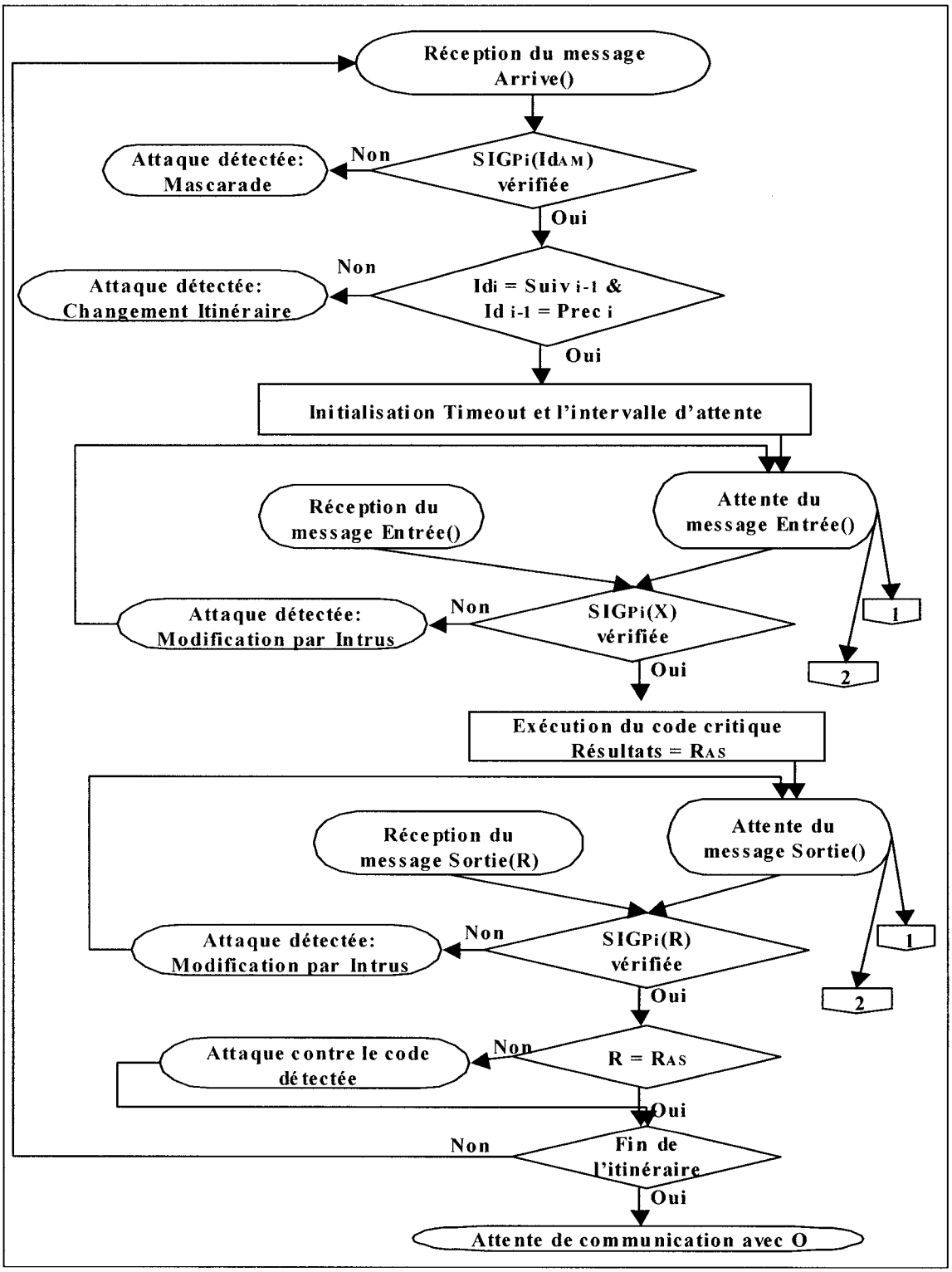

Figure 3.8 Algorithme suivi par l'agent sédentaire

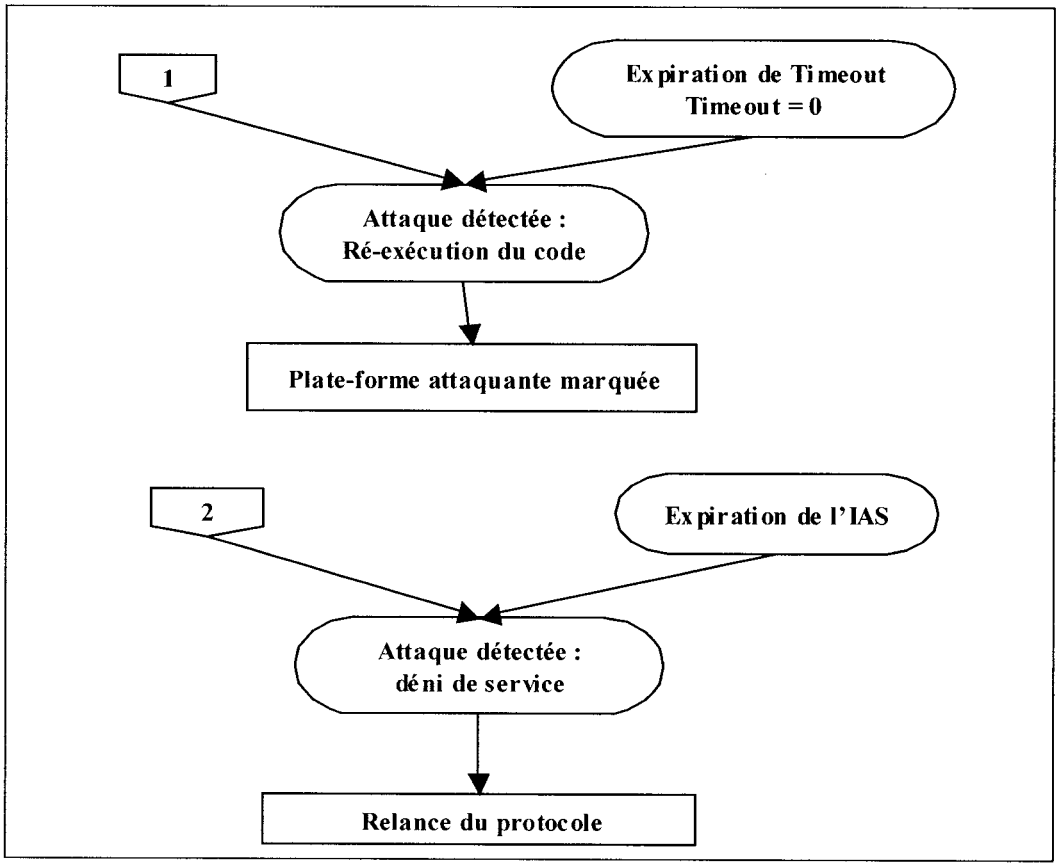

Figure 3.9 Algorithme suivi par l'agent sédentaire (Suite)

3.2.3 Dénouement du protocole

Une fois son exécution terminée à l'intérieur de la dernière plate-forme P_L, l'agent mobile migre vers la plate-forme d'origine O. Celle-ci contacte l'agent sédentaire qui lui envoie la liste des plates-formes attaquantes et les différents résultats obtenus, puis termine son exécution à l'intérieur de la tierce plate-forme T. La plate-forme d'origine O valide les résultats obtenus dans chaque plate-forme visitée. Les résultats des plates-formes marquées attaquantes par l'agent sédentaire sont éliminés de la liste des résultats de l'agent mobile. La plate-forme d'origine O met à jour sa liste des plate-formes attaquantes pour qu'elle puisse les éviter lors des prochaines exécutions des agents mobiles.

L'objectif de notre protocole est de détecter les différentes attaques déterminées

au début de ce chapitre. Une fois l'attaque détectée, le concepteur d'une application d'agents mobiles qui implémente le protocole peut trancher sur le comportement de son système suivant les besoins de son application. Cependant, nous suggérons de ne relancer le protocole qu'après une attaque de déni de service, une mascarade ou bien une attaque de changement d'itinéraire. Après les autres attaques, l'agent sédentaire *AS* marque la plate-forme comme étant attaquante et laisse l'agent mobile *AM* continuer son itinéraire. Ces choix permettrons à l'agent mobile de garder les résultats obtenus à l'intérieur des plates-formes non-attaquantes quand cela est possible.

3.3 Sécurité du protocole

Après avoir décrit le fonctionnement de notre protocole et les échanges qui sont effectués entre l'agent mobile et son agent coopérant, nous allons expliciter les différents scénarios d'attaques d'une plate-forme malveillante et la façon dont notre protocole permet de détecter ces attaques et de protéger l'agent mobile.

Soit P_i (i entre 1 et L) une plate-forme malveillante qui tente d'attaquer l'agent mobile. Les scénarios d'attaque de cette plate-forme se résument de la manière suivante:

1- La plate-forme P_i modifie les messages de type *Entrée()* et *Sortie()* qui sont transmis de l'agent mobile vers son agent sédentaire ;

2- La plate-forme P_i modifie le code critique de l'agent mobile ;

3- La plate-forme P_i retourne des résultats incorrects des appels systèmes effectués à l'intérieur du code critique ;

4- La plate-forme P_i exécute le code critique de l'agent mobile d'une manière incorrecte ;

5- Un intrus intercepte les trois messages transmis de l'agent mobile vers son agent sédentaire ;

6- La plate-forme P_i modifie les résultats obtenus par une plate-forme P_j tel que $j<i$;

7- La plate-forme P_i analyse le code de l'agent mobile et tente de le copier et de le ré-exécuter ou bien elle effectue un déni de service ;

8- La plate-forme P_i modifie l'itinéraire de l'agent mobile.

1- La plate-forme P_i modifie les messages de type Entrée() et Sortie() qui sont transmis de l'agent mobile vers son agent sédentaire.

Dans ce scénario, trois cas de modification de messages se présentent :

a- P_i change uniquement le message *Entrée()* ;

b- P_i change uniquement le message *Sortie()* ;

c- P_i change les deux messages *Entrée()* et *Sortie()* et ré-exécute le code.

Dans le premier cas, la plate-forme P_i modifie le message de type *Entrée()* et laisse le message de type *Sortie()* inchangé. Plus en détails, la plate-forme P_i exécute la partie du code critique de l'agent mobile avec comme entrée des données X, elle modifie le message *Entrée()* en remplaçant les données X par d'autre données X', puis elle transmet le message *Sortie()* avec les même résultats R obtenus. La Figure 3.10-a illustre ce scénario.

L'agent sédentaire, quant à lui, reçoit le message Entrée(), extrait les données modifiées X', exécute le code critique avec les données X', obtient les résultats R_{AS}, compare ceux-ci avec les résultats R et constate qu'ils sont différents. Ainsi, l'agent sédentaire découvre que la plate-forme P_i est une plate-forme attaquante. Dans le deuxième cas, la plate-forme malveillante P_i modifie le message de type *Sortie()* et laisse le message de type *Entrée()* inchangé. Plus précisément, la plate-forme transmet le message *Entrée()* qui contient des données X tel quel, elle exécute la partie du code critique et remplace les résultats R obtenus par d'autres résultats R', avant de transmettre le message *Sortie()*. La Figure 3.10-b illustre ce scénario. Dès que l'agent sédentaire reçoit le message *Entrée()*, il extrait les données X, exécute la partie du code critique pour obtenir les résultats R_{AS} et compare ceux-ci avec les résultats transmis R' qui ne sont pas concordants. Ainsi, l'agent sédentaire réalise que la plate-forme P_i est une plate-forme attaquante.

Dans le troisième cas illustré à la Figure 3.10-c, la plate-forme P_i modifie le message de type *Entrée()* avant de le transmettre, en remplaçant les données X par d'autre données X'. Puis, elle ré-exécute le code critique ou sa copie et modifie le

message de type *Sortie()* avant de le transmettre en remplaçant les résultats *R* obtenus initialement par des résultats *R'* obtenus lors de la deuxième exécution du code critique. Quand l'agent sédentaire reçoit le message *Entrée()*, il extrait les données modifiées *X'*, il exécute le code critique et obtient les résultats R_{AS} qui sont concordants avec les résultats extraits *R'* contenus dans le message *Sortie()*. Ainsi, l'agent sédentaire ne se rendra pas compte que la plate-forme P_i a attaqué l'agent mobile. Une fois retourné chez sa plate-forme d'origine *O*, l'agent mobile compare ses résultats avec ceux enregistrés dans l'agent sédentaire. Le système détectera cette attaque, mais il ne pourra pas savoir laquelle des plates-formes a attaqué. Les résultats obtenus dans la plate-forme P_i peuvent être modifiés en cours de route dans une plate-forme P_j tel que $i<j$.

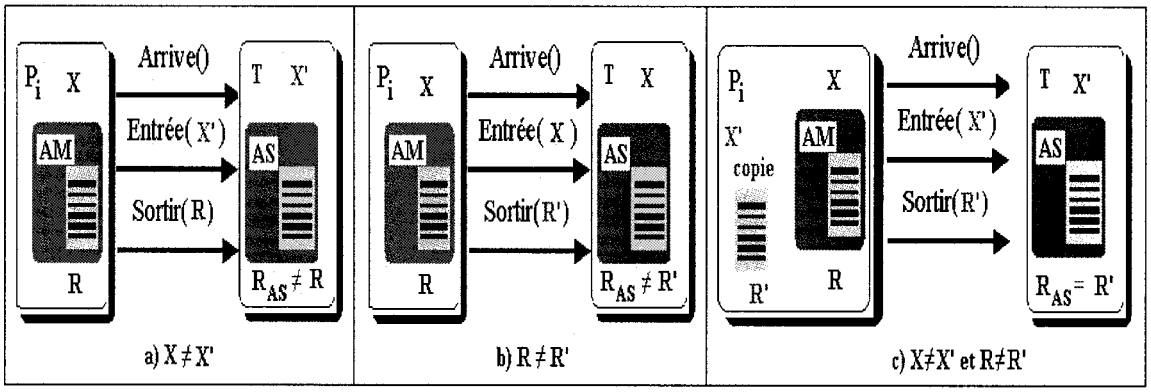

Figure 3.10 Premier scénario d'attaque

2- La plate-forme P_i modifie le code critique de l'agent mobile.

Deux cas se présentent dans ce scénario :

a- P_i modifie le code avant de l'exécuter ;

b- P_i modifie le code après son exécution.

Le premier cas est illustré à la Figure 3.11-a. La plate-forme P_i modifie le code critique avant de l'exécuter. Elle transmet les messages de type *Entrée()* et *Sortie()* sans modifier leur contenu. L'agent sédentaire extrait les données *X* du message *Entrée()*, il exécute le code critique et compare ses résultats R_{AS} avec les résultats R_i contenus dans le message *Sortie()* reçu; il réalise que les deux résultats sont différents. Ainsi, il conclut que la

plate-forme P_i est une plate-forme attaquante. Puisque le code de l'agent est modifié par la plate-forme P_i, il le sera dans toute les autres plates-formes P_j tel que $i<j$. L'agent sédentaire marquera toutes ces plates-formes P_j ($i<j$) comme étant attaquante, alors qu'elles ne le sont pas. Quand l'agent mobile retournera à sa plate-forme d'origine O, celle-ci détectera la modification du code de l'agent et, par conséquent, elle peut reconnaître la plate-forme qui a modifié le code de l'agent mobile. Dans ce cas, la plate-forme d'origine relance le protocole en omettant la plate-forme P_i de l'itinéraire de l'agent mobile.

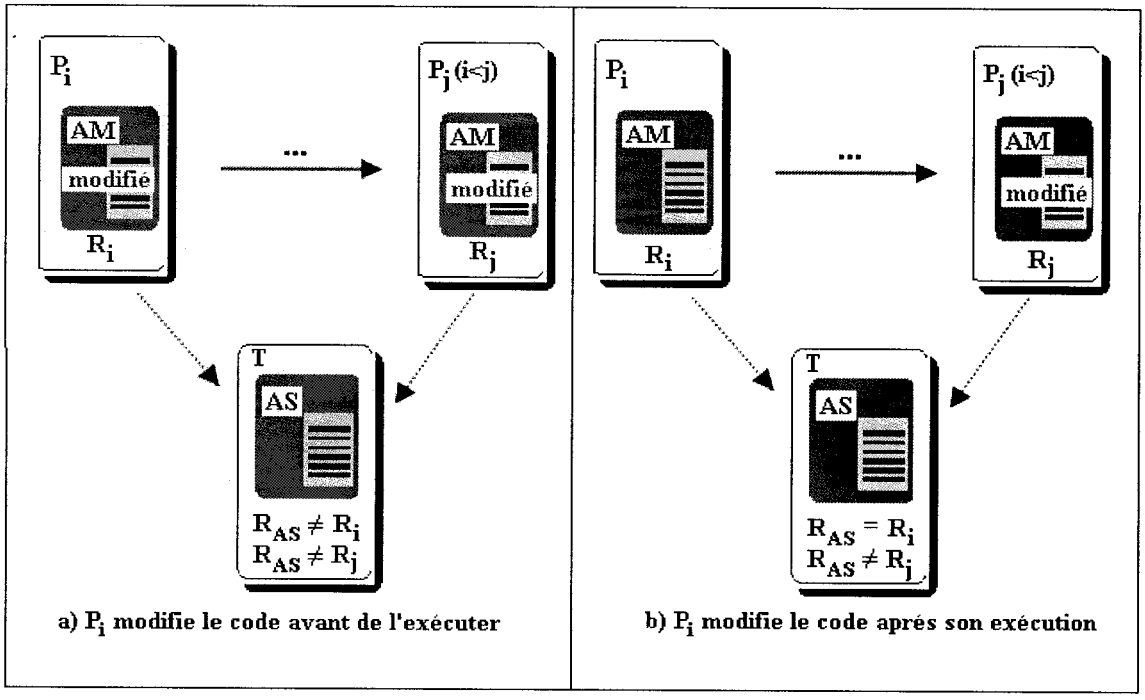

Figure 3.11 Deuxième scénario d'attaque : modification du code

Le deuxième cas est similaire au premier, sauf que la plate-forme P_i,dans ce cas, modifie le code critique après l'exécution de celui-ci. La Figure 3.11-b illustre ce scénario. Dans ce cas aussi, l'agent sédentaire marquera toutes les plates-formes P_j tel que $i<j$ comme attaquantes. De la même manière, une fois l'agent mobile est retourné à sa plate-forme d'origine O, cette dernière vérifie le code de l'agent mobile et réalise que

la plate-forme attaquante est soit P_i soit P_{i+1}. En effet, la plate-forme ne pourra pas savoir si c'est la plate-forme P_i qui a modifié le code critique de l'agent mobile après son exécution ou bien c'est la plate-forme P_{i+1} qui a modifié le code critique avant son exécution. Les deux suppositions donnent le même résultat.

3- La plate-forme P_i retourne des résultats incorrects des appels systèmes effectués à l'intérieur du code critique.

Dans ce scénario, la plate-forme P_i retourne des résultats incorrects quand l'agent mobile fait des appels systèmes. À titre d'exemple, la plate-forme P_i retourne une fausse valeur de temps du système ou bien retourne une fausse valeur de l'identité de la plate-forme où l'agent mobile est entrain de s'exécuter. Si ces appels système ont une influence sur le résultat de l'exécution de la partie du code critique de l'agent mobile, notre protocole permet de détecter cette attaque, vu que le résultat obtenu par l'agent sédentaire sera différent de celui obtenu par l'agent mobile. Cependant, le protocole est incapable de détecter une attaque où la plate-forme P_i retourne un résultat incorrect d'un appel système qui n'influence pas les résultats de l'exécution de la partie du code critique de l'agent mobile. L'agent sédentaire ne pourra pas détecter cette attaque, vu qu'il ne verra pas une différence entre ces résultats et les résultats obtenus par l'agent mobile.

4- La plate-forme P_i exécute le code critique de l'agent mobile d'une manière incorrecte.

Dans ce quatrième scénario, la plate-forme P_i exécute la partie du code critique de l'agent mobile d'une manière incorrecte et n'effectue aucune modification des messages de type *Entrée()* et *Sortie()* échangés entre l'agent mobile et son agent sédentaire. Plus en détails, la plate-forme P_i envoie à l'agent sédentaire le message de type *Entrée()* contenant les données X nécessaires à l'exécution du code critique de l'agent mobile, exécute le code critique d'une manière incorrecte en retournant à titre d'exemple la valeur *Faux* au lieu de la valeur *Vraie* en exécutant une instruction de

comparaison de son offre avec une offre minimale. Puis, elle envoie le message de type *Sortie()* contenant les résultats R de son exécution à l'agent sédentaire. Après avoir reçu le message *Entrée()*, l'agent sédentaire extrait les données X, il exécute la partie du code critique d'une manière correcte puisque la plate-forme T est supposée effectuer des exécutions correctes. Il obtient des résultats R_{AS} correctes. Quand il compare les deux résultats R et R_{AS}, il réalise que la plate-forme P_i est une plate-forme attaquante.

5- Un intrus intercepte les trois messages transmis de l'agent mobile vers son agent sédentaire.

Dans ce scénario, la plate-forme P_i est supposée effectuer tout le traitement d'une manière juste; elle exécute correctement la partie du code critique de l'agent mobile et envoie les trois messages *Arrive()*, *Entrée()* et *Sortie()* sans modifier leur contenu. Cependant, un intrus, un tierce élément malveillant différent de la plate-forme P_i et de la plate-forme T, intercepte les trois messages transités entre les deux plates-formes P_i et T. L'objectif de l'intrus est de pouvoir espionner les données envoyées dans les trois messages en vue de les modifier ou de les utiliser d'une manière frauduleuse.

Si l'intrus intercepte le message de type *Entrée()* contenant Id_i, $SIG_{Pi}(X)$ et X ou bien de type *Sortie()* contenant Id_i, $SIG_{Pi}(R)$ et R, il ne peut pas espionner leur contenu, vu que ces deux types de message sont cryptés par la clé publique de la plate-forme T qui est la seule à pouvoir les décrypter avec sa clé privée. Ainsi, l'intrus ne pourra jamais espionner les données sensibles envoyées par l'agent mobile vers son agent coopérant. L'intrus ne peut forger les deux types de message sans que l'agent sédentaire ne détecte cette modification, vu que les données envoyées sont signées par la clé privée de la plate-forme P_i. De cette manière, notre protocole assure la confidentialité et l'intégrité des données transmises de l'agent mobile vers l'agent sédentaire.

6- La plate-forme P_i modifie les résultats obtenus par une plate-forme P_j tel que $j<i$.

Dans ce scénario, la plate-forme P_i tente de modifier les résultats R_j obtenus par l'agent mobile à l'intérieur d'une plate-forme P_j qui précède la plate-forme P_i. Nous

supposons que la plate-forme P_i a un accès en lecture/écriture des données transportées par l'agent mobile. La plate-forme P_i tente, par exemple, d'augmenter l'offre de son concurrent P_j pour qu'elle soit plus élevée que son offre. Cette attaque est détectée quand l'agent mobile retourne à sa plate-forme d'origine O. À chaque étape d'exécution de l'agent mobile à l'intérieur d'une plate-forme P_k (k entre 1 et L), l'agent sédentaire enregistre les résultats obtenus par l'agent mobile d'une manière sécuritaire, vu que la plate-forme T est supposée être fiable et ne collabore pas avec une plate-forme contre une autre. Les résultats contenus dans l'agent sédentaire demeurent correctes le long de l'itinéraire de l'agent mobile. Dés que l'agent mobile retourne chez lui, sa plate-forme d'origine O compare ses résultats avec ceux enregistrés dans l'agent sédentaire. La plate-forme O détecte que les résultats R_j obtenus dans la plate-forme P_j sont modifiés et réalise qu'une attaque a eu lieu. Cependant, elle ne peut trancher sur l'identité de la plate-forme qui a modifié frauduleusement ces résultats. La plate-forme attaquante peut être une plate-forme Pi avec i entre j et L ($j \leq i \leq L$).

7- La plate-forme P_i analyse le code de l'agent mobile, tente de le copier et de le ré-exécuter, ou bien elle effectue une attaque de déni de service.

Dans ce scénario, quand l'agent mobile arrive à la plate-forme P_i, celle-ci tente d'analyser son code afin de le copier et de le ré-exécuter, ou bien de connaître la stratégie d'exécution de l'agent mobile. Notre protocole permet de remédier à une telle situation en la détectant et en agissant à temps. L'agent sédentaire initialise un compteur de temps *Timeout* avec une valeur limitant le temps d'exécution à l'intérieur de la plate-forme P_i. Cette valeur initiale de *Timeout* présente le temps maximum nécessaire à l'exécution de l'agent mobile dans des conditions normales où la plate-forme P_i exécute le code de l'agent mobile sans tenter de l'attaquer. Or, les opérations d'analyse, de duplication et de ré-exécution du code nécessitent un temps additionnel au temps d'exécution normale du code. Donc, si la plate-forme P_i tente cette attaque, la valeur de *Timeout* expire et, par conséquent, l'agent sédentaire détecte l'attaque et marque la plate-forme P_i comme étant une plate-forme attaquante.

La plate-forme P_i peut aussi tenter d'attaquer l'agent mobile avec le déni de service, i.e. le tuer ou retarder simplement son exécution. Cette attaque causerait beaucoup de dommages dans le champ des applications qui sont sensibles à la durée d'exécution. Pour détecter cette attaque et la distinguer de la première, l'agent sédentaire attend l'arrivée du message *Sortie()* pendant un autre intervalle de temps d'attente. Après l'écoulement de cette dernière durée, l'agent sédentaire détecte l'attaque de déni de service et agit en conséquence.

L'intérêt d'ajouter un autre intervalle d'attente après l'expiration de *Timeout* est justifié par la distinction entre l'attaque déni de service et les autres attaques où une plate-forme malveillante tentera d'analyser le code de l'agent mobile afin de le copier et de le ré-exécuter. Cette distinction permet à l'agent sédentaire de connaître la nature de l'attaque et d'agir en fonction de celle-ci. En effet, dans le cas d'une attaque autre que le déni de service l'agent sédentaire marque seulement la plate-forme comme une plate-forme attaquante et laisse l'agent mobile continuer son chemin. Ainsi, le mécanisme ne se réinitialise qu'après une attaque de déni de service où la plate-forme attaquante tentera de tuer l'agent mobile ou de retarder son exécution.

8- La plate-forme P_i modifie l'itinéraire de l'agent mobile.

Vu que notre protocole s'inspire du protocole de Roth [ROT98] pour détecter les attaques visant à modifier l'itinéraire de l'agent mobile, le dernier scénario est lui aussi inspiré de celui proposé par Roth. La plate-forme P_i tente d'envoyer l'agent mobile sur une plate-forme P'_i différente de la plate-forme P_{i+1} choisie par l'agent mobile. Les éléments permettant d'enregistrer l'itinéraire de l'agent mobile par l'agent sédentaire sont envoyés dans les messages de type *Arrive()* et *Sortie()*. L'identité de la plate-forme courante Id_i ainsi que l'identité $Prec_i$ de la plate-forme précédente d'où provient l'agent mobile sont envoyées dans le message *Arrive()*. Quant à l'identité $Suiv_i$ de la plate-forme où l'agent mobile veut migrer, elle est envoyée dans le message de type *Sortie()*.

Trois cas se présentent dans ce scénario:

a) la plate-forme attaquante P_i ne modifie pas la valeur de $Suiv_i$ et la plate-forme P'_i ne procède pas à une mascarade ;

b) la plate-forme P_i ne modifie pas la valeur de $Suiv_i$ mais la plate-forme $P'i$ effectue une mascarade ;

c) la plate-forme change la valeur de $Suiv_i$.

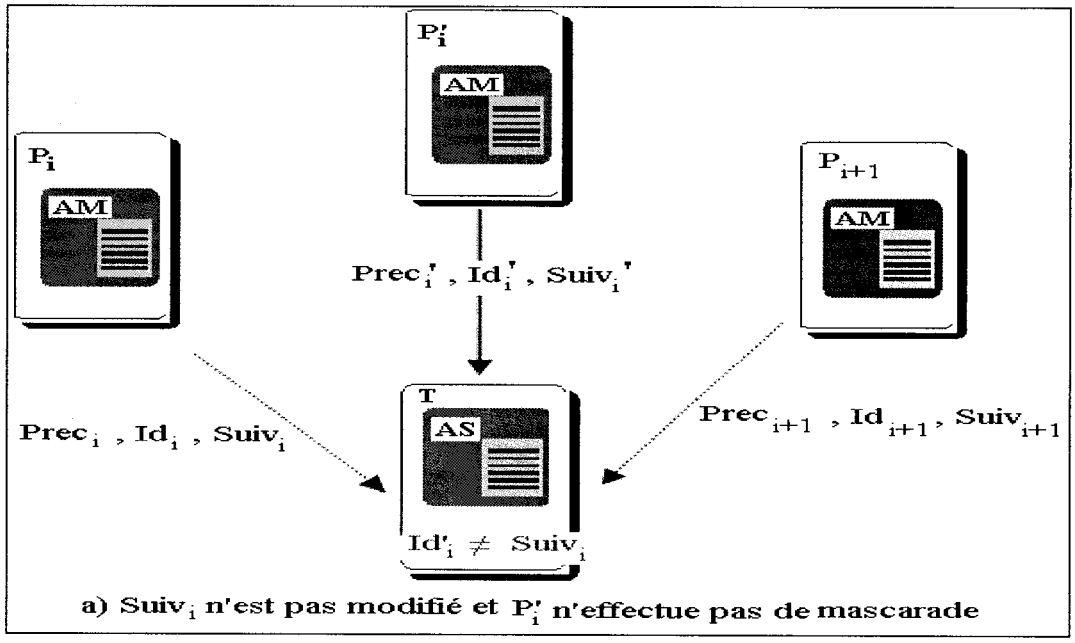

Figure 3.12 Attaque : changement d'itinéraire (Premier cas)

La Figure 3.12 illustre le premier cas du dernier scénario. La plate-forme P_i envoie le message *Sortie()* généré par l'agent mobile tel quel, i.e. sans changer la valeur de $Suiv_i$ contenant l'identité Id_{i+1} de la plate-forme P_{i+1} où l'agent mobile veut migrer. Puis, elle envoie l'agent mobile vers la plate-forme P'_i au lieu de l'envoyer à la plate-forme P_{i+1}. Une fois que l'agent mobile arrive à la plate-forme P'_i, celle-ci envoie à l'agent sédentaire le message *Arrive()* sans effectuer une mascarade, i.e. sans remplacer son identité Id'_i par celle de la plate-forme P_{i+1}. L'agent sédentaire vérifie les égalités de la validation de l'itinéraire de l'agent mobile. Il trouve que l'égalité $Id'_i = Suiv_i$ n'est pas

vérifiée et par conséquent il détecte l'attaque et marque la plate-forme P_i comme une plate-forme attaquante.

Dans le deuxième cas qui est illustré par la Figure 3.13, la plate-forme P_i envoie le message *Sortie()* tel quel, i.e. sans changer la valeur de $Suiv_i$. Puis, elle envoie l'agent mobile à la plate-forme P'_i au lieu de la plate-forme P_{i+1}. La plate-forme P'_i effectue une mascarade en envoyant dans le message *Arrive()* l'identité de la plate-forme P_{i+1} au lieu de son identité. Quand l'agent sédentaire reçoit le message *Arrive()*, il vérifie l'égalité de la validation de l'itinéraire de l'agent mobile $Suiv_i = Id'_i$ puisque $Id'_i = Id_{i+1}$, mais la vérification de la signature de la plate-forme P'_i sur l'identité de l'agent mobile génère une erreur puisque la signature est calculée à l'aide de la clé privée de la plate-forme visitée. Ainsi, l'agent sédentaire détecte cette attaque et marque la plate-forme P_i comme une plate-forme attaquante.

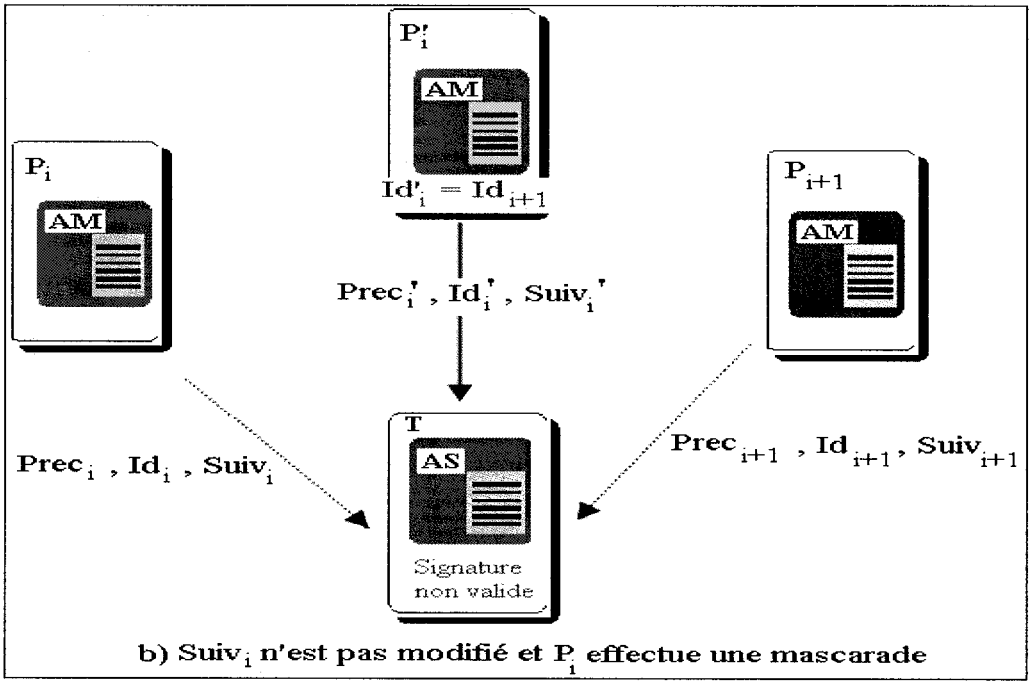

Figure 3.13 Attaque : changement d'itinéraire (Deuxième cas)

Dans le dernier cas de ce scénario d'attaque et à l'envoi du message de type

Sortie(), la plate-forme P_i remplace l'identité de la plate-forme P_{i+1} par l'identité de la plate-forme P'_i. Elle envoie *Sortie(*$Suiv_i$=Id'_i, ...*)* au lieu de *Sortie(*$Suiv_i$=Id_{i+1}, ...*)*. Puis, elle envoie l'agent mobile vers la plate-forme P'_i. La Figure 3.14 illustre ce scénario. Quand l'agent mobile arrive à la plate-forme P'_i, celle-ci envoie le message *Arrive()* d'une manière ordinaire. L'agent sédentaire vérifie les égalités de la validation de l'itinéraire de l'agent mobile et la signature de la plate-forme P'_i. Il trouve que toutes les vérifications sont correctes. P'_i envoie le message *Sortie()* avec $Suiv_i = Id_{i+1}$ et envoie l'agent mobile à la plate-forme P_{i+1}. De cette manière, l'agent coopérant ne peut pas détecter cette attaque. Quand l'agent mobile retourne à sa plate-forme d'origine O, il peut vérifier l'itinéraire qu'il a enregistré avec l'itinéraire enregistré par l'agent sédentaire. L'agent mobile peut détecter la modification de son itinéraire en faisant la comparaison de l'itinéraire qu'il a enregistré avec celui enregistré par son agent sédentaire, et il ignore les résultats obtenus à l'intérieur de la plate-forme P_i.

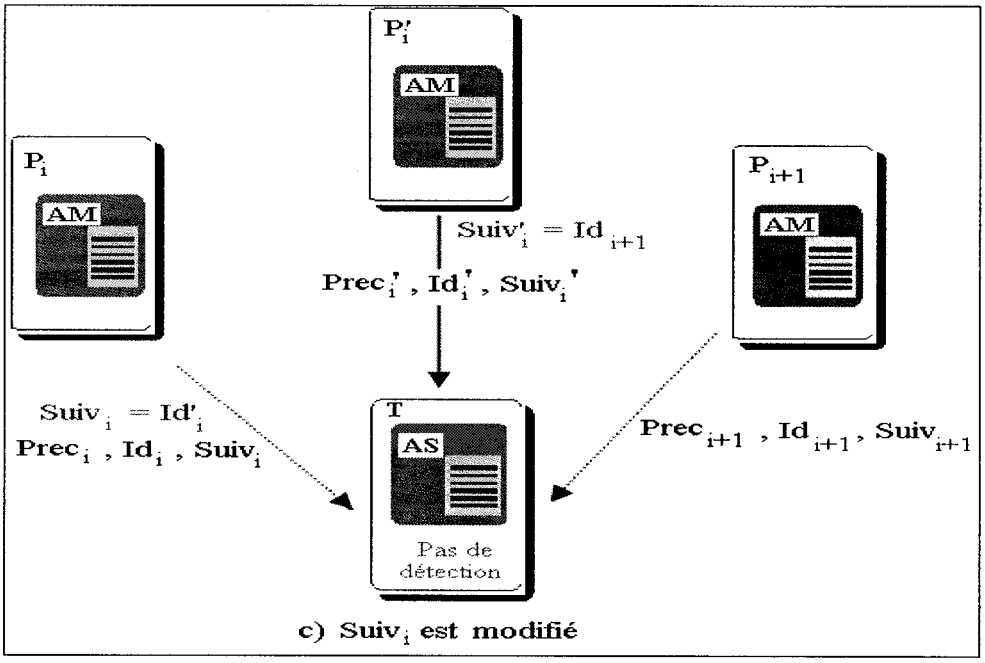

Figure 3.14 Attaque : changement d'itinéraire (Troisième cas)

3.4 Attaques non détectées par notre protocole

La protection du code contre la modification et les exécutions incorrectes est basée sur sa duplication et son exécution par les deux agents mobile et sédentaire. Les attaques sont détectées en comparant les résultats des deux agents. De ce fait, le code critique qui ne retourne pas de résultats ne peut être protégé par ce mécanisme. Une solution à ce problème est de calculer la trace cryptographique [VIG98] du code critique qui constituera les résultats des exécutions du code.

Dans le cas où une plate-forme (P_i) attaquante change les deux messages *Entrée()* et *Sortie()* et exécute une copie du code, l'agent sédentaire ne peut pas détecter cette attaque, vu que ses résultats seront concordants avec ceux envoyés dans le message *Sortie()* modifié. Cependant, l'attaque sera détectée par la plate-forme d'origine O quand l'agent mobile retournera chez lui. Mais, elle ne peut pas reconnaître la plate-forme attaquante parce que les résultats obtenus à l'intérieur de la plate-forme P_i peuvent être modifiés par une plate-forme P_j tel que $i \leq j \leq L$. Une solution à ce problème est de protéger les résultats transportés par l'agent mobile à l'aide d'un mécanisme cryptographique comme celui implémenté par le système Ajanta [KAR00] où les résultats sont stockés dans un tableau à ajout seulement.

L'agent sédentaire ne détecte pas l'attaque de changement d'itinéraire lorsqu'une plate-forme P_i change la valeur de $Suiv_i$ (voir le dernier scénario d'attaque) et la plate-forme P'_i, où l'itinéraire de l'agent mobile est dévié, exécute normalement le code de l'agent mobile et le renvoie à la plate-forme qu'il a choisie. Une solution à ce problème est d'enregistrer l'itinéraire par l'agent mobile dans un tableau à ajout seulement. Ainsi, la plate-forme d'origine O sera capable de comparer l'itinéraire enregistré par l'agent sédentaire avec celui enregistré par l'agent mobile.

3.5 Inconvénients et avantages

Le principal inconvénient de notre protocole est sa faible tolérance aux pannes due à l'utilisation de la tierce plate-forme fiable qui peut aussi constituer un goulot

d'étranglement. Un autre inconvénient réside dans la difficulté de calculer les valeurs initiales des compteurs temporels d'une manière adéquate. Par contre, le protocole a su utiliser les échanges de messages entre les deux agents mobile et coopérant sans dégrader les performances de l'agent mobile. Un autre point fort de notre protocole est qu'il a évité l'utilisation des fonctions cryptographiques et a pu simplifier la coopération de l'agent sédentaire.

3.5.1 Inconvénients

L'approche que nous proposons est basée sur l'hypothèse stipulant l'existence d'une tierce plate-forme fiable exécutant le code d'une manière correcte. Cette plate-forme est sollicitée par tous les agents mobiles qui sont actifs en même temps. De ce fait, cette plate-forme constitue le goulot d'étranglement de notre protocole. Ce problème devient de plus en plus critique quand le système est déployé à grande échelle dans un réseau étendu. Pour remédier à une telle situation, le système mettra à la disposition des plates-formes plusieurs tierces plates-formes fiables et implémentera un mécanisme de répartitions des demandes des plates-formes d'origine pour équilibrer les charges entre les différentes plates-formes.

Pour la même raison, notre protocole est considéré comme un système centralisé dont la fiabilité est réduite, vu que, si la tierce plate-forme tombe en panne, tout le mécanisme s'arrête. Pour contourner ce fléau que notre protocole partage avec les systèmes centralisés, l'agent sédentaire sauvegarde ses données d'une manière persistante afin de l'instancier correctement après une panne. Le système peut créer aussi une plate-forme miroir qui remplacera la tierce plate-forme en cas de panne prolongée.

Un autre problème auquel notre protocole fait face est celui d'utiliser des compteurs de temps *Timeout* et l'intervalle d'attente supplémentaire. Le choix des valeurs initiales de ces compteurs temporels doit se faire d'une manière judicieuse. La valeur initiale de *Timeout* présente le temps maximum nécessaire à l'exécution du code de l'agent mobile. Après l'expiration de *Timeout*, l'agent sédentaire conclut que la plate-

forme est attaquante. Si la valeur de *Timeout* est très petite, le système marquera toutes les plates-formes comme attaquantes et se trompera toujours sur la fiabilité des plates-formes visitées par l'agent mobile. Par contre, si la valeur de *Timeout* est très grande, le système ne pourra pas détecter des attaques d'analyse du code qui se font rapidement. Le même raisonnement s'applique aussi au temps d'attente supplémentaire qui permet à l'agent sédentaire de détecter l'attaque déni de service. Pour remédier à ce problème, le système peut doter l'agent sédentaire des mécanismes qui lui permettront de calculer ces valeurs initiales d'une manière dynamique et en fonction des conditions variables des environnements d'exécution. À chaque étape d'exécution de l'agent mobile, l'agent sédentaire calculera les valeurs initiales en fonction de la charge du réseau et de la charge à l'intérieur de la plate-forme exécutant l'agent mobile.

3.5.2 Avantages

Notre protocole utilise des échanges de messages entre l'agent mobile et un agent sédentaire afin d'assurer la sécurité des exécutions de l'agent mobile. Mais, même avec ces interactions, le mécanisme ne restreint pas les capacités de l'agent mobile en terme d'autonomie qui constitue un avantage de cette technologie. En effet, l'agent mobile prend ses décisions indépendamment de l'agent sédentaire dont le rôle est de participer à la sécurisation du système. De plus, ces échanges de messages ne pourront pas augmenter considérablement le temps de réponse des exécutions de l'agent mobile, vu que celui-ci n'attend jamais un message retourné par son agent coopérant et que, même si le système détecte une attaque, il ne fait que marquer la plate-forme attaquante au lieu de réinitialiser le mécanisme.

D'après l'analyse des approches protégeant l'agent mobile contre les plates-formes malveillantes, expliquée dans le deuxième chapitre, nous avons pu conclure que la meilleure façon de protéger le code de l'agent mobile est d'utiliser les fonctions cryptographiques [SAN98], mais l'inconvénient majeur de cette approche est qu'elle augmente la vulnérabilité des plates-formes exécutant un code crypté pouvant comporter des instructions malveillantes. Notre protocole évite l'utilisation de ces fonctions

cryptographiques et permet de détecter des attaques contre le code sans avoir recours à ces mécanismes.

Un autre avantage de notre protocole est que l'utilisation d'une tierce plate-forme fiable exécutant l'agent coopérant permet de simplifier notre protocole et d'augmenter sa sécurité. En effet, d'une part, l'agent mobile n'est obligé de retenir qu'une seule adresse, celle de la plate-forme fiable, contrairement aux autres approches [ROT98] utilisant la coopération des agents mobiles et où l'agent coopérant se déplace lui aussi. Dans ces dernières approches, l'agent mobile doit connaître l'itinéraire de son agent coopérant, ce qui augmente la complexité des protocoles. D'autre part, dans ces approches, le système doit se soucier aussi de la sécurité de l'agent coopérant et des plates-formes qu'il visite. Alors que l'agent coopérant est censé augmenter la sécurité de l'agent mobile, il devient un fardeau supporté par le système, qui est évité dans notre protocole.

CHAPITRE IV
VALIDATION FORMELLE, IMPLÉMENTATION ET RÉSULTATS

Après avoir décrit les spécifications de notre protocole sécuritaire protégeant l'agent mobile contre les attaques des plates-formes malveillantes, nous avons utilisé une approche de validation formelle pour vérifier les propriétés liées à la détection des attaques. Nous avons modélisé le protocole avec les outils de model-checker SPIN et formalisé les propriétés avec la logique temporelle linéaire. Puis, nous avons implémenté le protocole dans un exemple d'agent simple pour s'assurer de ses fonctionnalités liées à la protection de l'agent mobile et pouvoir mesurer les coûts de l'implémentation en terme du trafic généré et du temps d'exécution. Dans ce chapitre, nous allons présenter le modèle formel de notre protocole, puis la manière dont nous avons vérifié ses propriétés et les résultats de la validation. Ensuite, nous allons décrire l'environnement de l'implémentation et les choix de mise en œuvre du protocole. Enfin, nous allons présenter les tests effectués pour évaluer cette implémentation, ainsi que les mesures prises et que nous avons analysées pour en déduire l'efficacité de notre protocole.

4.1 Validation formelle du protocole

Il existe plusieurs méthodes pour valider des protocoles de communication, les principales étant le test, la démonstration automatique et le *model-checking*. Le test est indispensable et permet de découvrir de nombreuses erreurs, mais il ne peut pas être exhaustif. La démonstration automatique [SCH99] permet de répondre à toutes les questions de la vérification mais sa mise en pratique est souvent lourde et compliquée. Le model-checking se veut un mécanisme entre les deux. Il s'agit d'une méthode exhaustive et en grande partie automatique. Le travail de vérificateur se limite à construire un modèle formel du système et à formaliser les propriétés à vérifier. Nous avons utilisé cette dernière méthode pour valider formellement notre protocole et vérifier

certaines de ses propriétés liées à la détection des attaques des plates-formes contre un agent mobile.

Nous avons choisi le model-checker SPIN comme outil pour l'implémentation de notre modèle formel vu qu'il est le plus approprié pour la vérification de notre protocole. En effet, il permet de simuler et de vérifier les algorithmes répartis. Il est disponible librement sur Internet [HOL90]. Pour être étudié, un système est d'abord décrit à l'aide de Promela, le langage de spécification de SPIN. Promela permet de décrire le comportement de chacun des processus du système et les interactions entre ces processus à travers des canaux de communication. Ensuite, SPIN permet de simuler des exécutions de système afin de se familiariser avec le comportement de celui-ci. Enfin, il permet de vérifier, par un parcours exhaustif de l'ensemble des états atteignables, que le système satisfait bien certaines propriétés exprimées par exemple en *LTL*, la *logique temporelle linéaire*.

4.1.1 Description du modèle formel du protocole

Le modèle formel est composé de deux processus *AgentMobile()* et *AgentSédentaire()* qui présentent le comportement de l'agent mobile et celui de l'agent sédentaire respectivement, et d'un canal de communication *AMtoAS* qui modélise les échanges de messages entre les deux agents. Le comportement d'une plate-forme attaquante est intégré dans le processus *AgentMobile()* qui peut choisir de passer par une attaque ou bien de passer par une exécution normale. Prenons comme exemple la partie du modèle en Promela suivante :

```
    if                      /* Choix de la plate-forme suivante */
    ::Suiv = 0              /*Choix de la plate-forme d'origine */
    ::Suiv = (ID + 1)         /* Choix d'une autre plate-forme */
#if ModifItin
    :: Suiv = Random        /* Attaque Changement d'itinéraire */
#endif
    fi;
```

Le processus *AgentMobile()* peut choisir, d'une manière équitable, entre le retour à la plate-forme d'origine *O*, le déplacement à la plate-forme suivante ou bien l'attaque de changement d'itinéraire. Les attaques peuvent être activées ou désactivées selon le modèle que nous voulons vérifier. Elles sont modélisées suivant les scénarios d'attaques décrits dans le chapitre III, avec une seule restriction de deux scénarios qui ne sont pas inclus, le cinquième et le sixième, qui sont liés à l'interception des messages par un intrus et à la modification des résultas intermédiaires obtenus dans une plate-forme visitée par l'agent mobile. Ces deux scénarios ne sont pas inclus dans le modèle vu qu'ils ne présentent pas une grande importance dans notre protocole et leur ajout compliquera le modèle sans un grand apport à la vérification des propriétés du protocole. La partie du code critique de l'agent mobile est modélisée par une fonction linéaire, $f(x) = x + n$, qui présente l'avantage d'être simple et de retourner une valeur après son exécution. La plate-forme suivante est choisie en incrémentant de un l'identité de la plate-forme courante et, lors d'une attaque de changement d'itinéraire, l'identité de la plate-forme suivante est choisie d'une manière arbitraire. L'annexe I contient la totalité de code Promela de modèle formel de notre protocole.

Afin de s'assurer du bon fonctionnement de notre modèle, nous avons exécuté des simulations en activant une attaque à la fois. Dans la section suivante, nous allons présenter quelques-unes de ces simulations. La première simulation présente l'attaque de changement des messages de type *Entrée()* et *Sortie()*. Elle concorde parfaitement avec le premier scénario d'attaque décrit dans le chapitre III. La plate-forme P_1 envoie le message *Entrée()* avec $X=10$, calcule le résultat $R=X+5$ et attaque l'agent mobile en modifiant le résultat dans le message *Sortie()*, i.e. elle remplace $R=15$ par $R'=25$. Dans la deuxième attaque, P_2 change la donnée en entrée $X= 10$ par $X'=20$ et change le résultat $R=15$ par le résultat $R'=25$. Le protocole continue son exécution dans les deux cas après que l'agent sédentaire marque les deux plates-formes attaquantes. La Figure 4.1 illustre cette simulation.

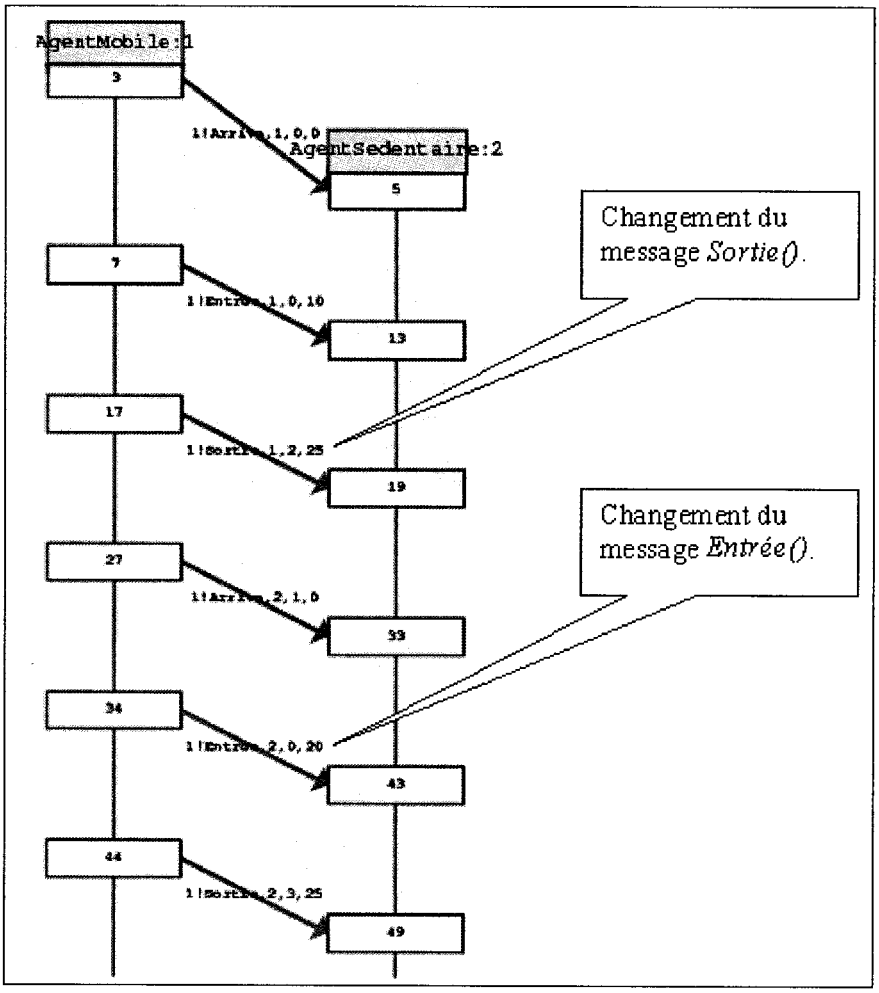

Figure 4.1 Simulation 1 : Changement des messages *Entrée*() et *Sortie*()

Pour comprendre les schémas générés par SPIN, nous présentons la sémantique de ses sorties.

Processus1 : Nom du processus.

N : Numéro d'un état dans une exécution du protocole.

→ : Envoi d'un message du processus *AgentMobile*() vers le processus *AgentSédentaire*(). La flèche est étiquetée par le type de message envoyé (Arrive, Entrée ou Sortie) avec leur paramètres.

N : Numéro d'état où l'exécution s'est arrêtée.

Il faut noter que les schémas n'illustrent qu'une partie des exécutions du protocole, celle qui aide à comprendre la validation.

La deuxième simulation présente l'attaque *déni de service* et elle concorde parfaitement avec le septième scénario d'attaque décrit dans le chapitre III. La plate-forme P_2 envoie le message de type *Entrée()* à l'agent sédentaire, mais elle s'abstient de lui envoyer le message de type *Sortie()*. L'agent sédentaire détecte l'attaque et arrête le protocole. La Figure 4.2 illustre cette simulation.

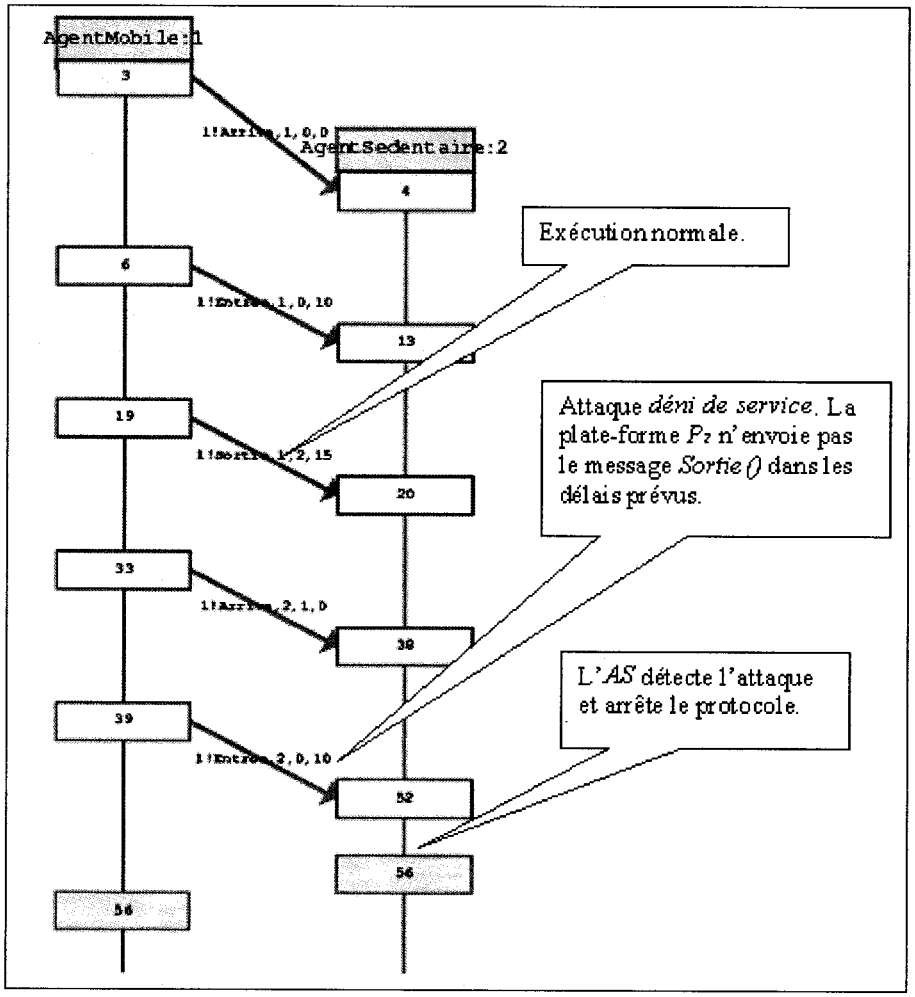

Figure 4.2 Simulation 2 : Déni de service

La dernière simulation présente l'attaque de changement d'itinéraire et concorde avec le scénario d'attaque 8-b décrit dans le chapitre III. La plate-forme P_1 envoie l'agent mobile à la plate-forme P_{100} au lieu de l'envoyer à la plate-forme P_2 qui doit être la plate-forme suivante. La plate-forme P_1 ne change pas la valeur de $Suiv_1$ dans le message de type *Sortie()* qui contient l'identité de la plate-forme où l'agent mobile veut aller. L'agent sédentaire détecte cette attaque et relance le protocole. La Figure 4.3 illustre cette simulation.

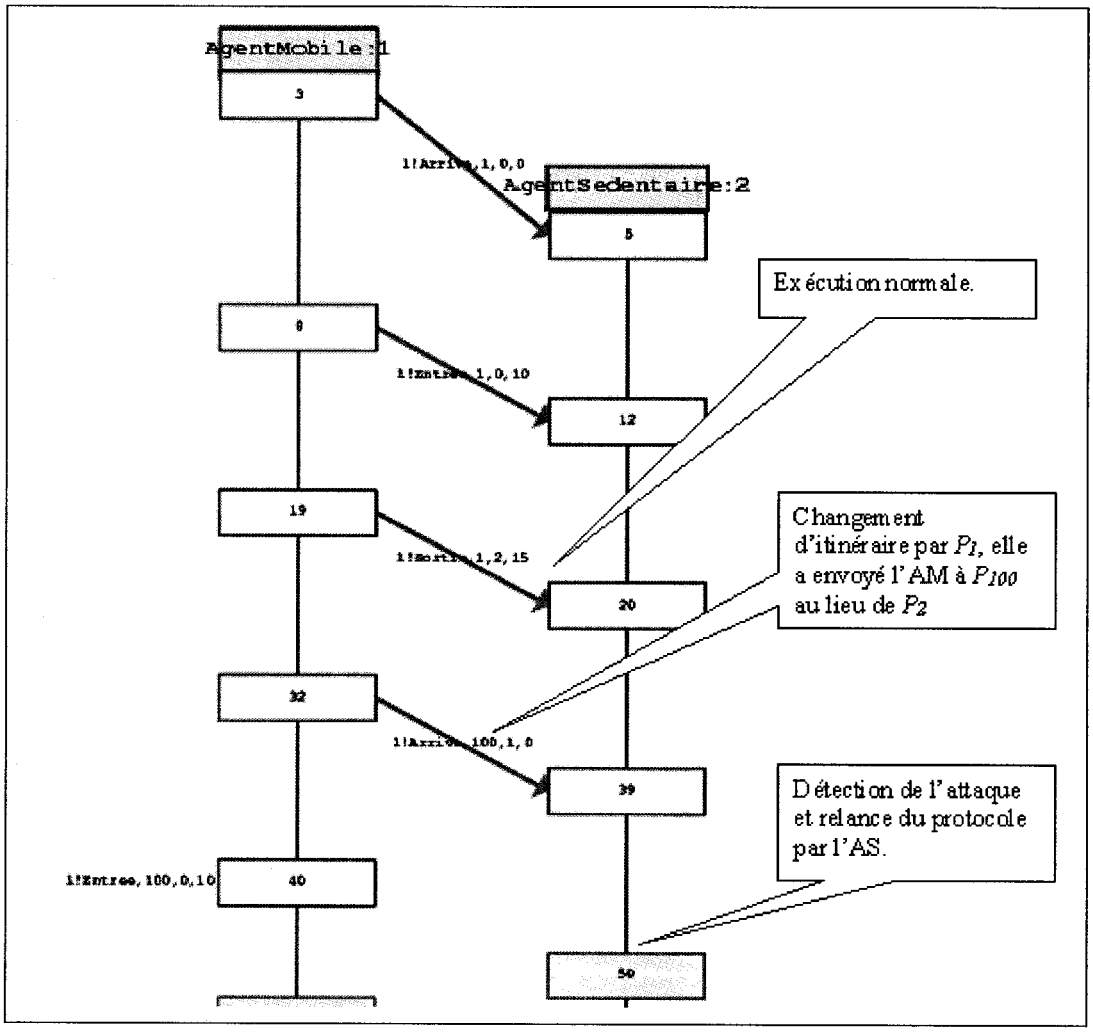

Figure 4.3 Simulation 3 : Changement d'itinéraire sans modification de message de type *Sortie()*

4.1.2 Vérification des propriétés du protocole

Après avoir modélisé le protocole en utilisant le langage Promela et s'être familiarisé avec son comportement en exécutant des simulations, nous avons vérifié deux types de propriétés du système : celles qui sont générales et celles qui sont liées à la détection des attaques des plates-formes malveillantes. Les différentes propriétés vérifiées sont :

1. l'absence du *blocage* (*deadlock*) ;
2. l'absence des *divergences* (*livelock*) ;
3. l'*AS* identifie correctement l'identité de la plate-forme attaquante ;
4. l'*AS* détecte toutes les attaques effectuées par une plate-forme malveillante ;
5. l'*AS* ne déclare jamais une détection tant qu'il n'y pas une attaque.

1. L'absence du blocage (deadlock)

L'absence de blocage est une propriété énonçant qu'un système ne se retrouve jamais dans une situation où il lui est impossible de progresser. Le model-checker SPIN détecte un blocage quand il arrive à la fin d'une séquence d'exécution sans atteindre un état final du système. Afin de marquer un état comme étant un état final normal, l'étiquette *end* est ajoutée au modèle Promela avant l'instruction signalant la fin d'un processus. Dans notre modèle, nous avons ajouté l'étiquette *end* à la fin du processus *AgentSédentaire()*. Une séquence d'exécution qui n'atteint pas cette étiquette est considérée comme un blocage. La vérification du modèle sans attaque et celui où toutes les attaques sont actives démontre que notre protocole ne contient aucune erreur de blocage.

2. L'absence de divergences (livelock) ou cycles sans progression

Un cycle sans progression présente le fait que le modèle exécute une boucle infinie sans passer par un état de progression. Afin de marquer un état comme étant un

état de progression, l'étiquette *progress* est ajoutée au modèle Promela avant l'instruction qui doit être exécutée par le protocole. Un cycle infini qui ne passe pas à travers cette étiquette est considéré comme un cycle sans progression. Dans notre modèle, nous avons ajouté l'étiquette *progress* au début du processus *AgentMobile()*. Un cycle infini qui ne passe pas par cette étiquette, i.e. n'envoie pas le message de type *Arrive()*, est considéré comme un cycle sans progression. La vérification de cette propriété des deux modèles, avec attaque et sans attaque, révèle que notre protocole ne contient aucun cycle sans progression.

3. L'AS identifie correctement l'identité de la plate-forme attaquante

Avec cette troisième propriété, nous voulons vérifier si l'agent sédentaire *AS* identifie correctement l'identité de la plate-forme attaquante. Pour cela, nous avons ajouté deux variables *Attaquante* et *Détectée*. La variable *Attaquante* contient l'identité de la plate-forme attaquante et la variable *Détectée* contient l'identité de la plate-forme détectée par l'agent sédentaire *AS*. Nous avons utilisé l'instruction ***assert(Attaquante == Détectée)*** qui permet de vérifier à chaque exécution que les deux identités des plates-formes attaquante et détectée sont égales. La vérification est effectuée en activant une attaque à la fois afin de valider la propriété pour chaque type d'attaque. La vérification de cette propriété démontre que notre protocole ne contient aucune erreur pour toutes les attaques sauf dans un seul scénario. La Figure 4.4 illustre l'exécution qui a causé cette erreur de validation. La plate-forme P_{254} attaque le code de l'agent mobile en le modifiant d'une façon permanente. La plate-forme P_{255} n'effectue aucune attaque mais elle exécute un code modifié. L'agent sédentaire *AS* se trompe sur la fiabilité de celle-ci. Ce comportement menant à cette erreur de validation est prédit dans le scénario d'attaque 2-b détaillé dans le chapitre III.

4. L'AS détecte toutes les attaques effectuées par une plate-forme malveillante

Avec cette quatrième propriété, nous voulons vérifier si l'agent sédentaire *AS* détecte tous les types d'attaques qu'une plate-forme malveillante effectue. Nous avons

formalisé cette propriété avec la logique temporelle linéaire *LTL* (*Linear Temporal Logic*). Cette logique temporelle permet d'exprimer formellement les propriétés du protocole et la façon dont elles évoluent dans le temps. Nous avons ajouté deux propositions atomiques à notre modèle, la proposition atomique *AttaqueEffectuée* qui présente le fait qu'une plate-forme a attaqué l'agent mobile et la proposition atomique *AttaqueDétectée* qui présente le fait que l'agent sédentaire *AS* a détecté cette attaque. Nous avons utilisé les trois combinateurs de la logique LTL : *toujours* [], *inévitablement* ◊, et *implication* ⇒.

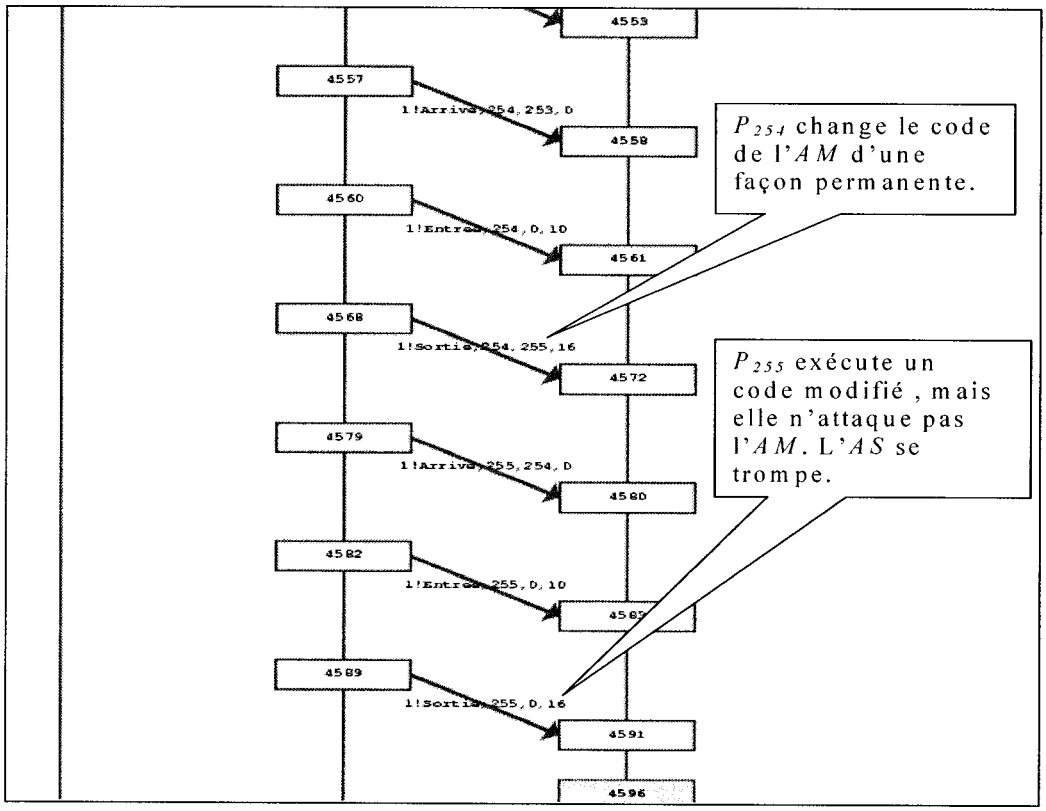

Figure 4.4 Explication de l'erreur de validation de la troisième propriété

Soit les deux propositions atomiques P et Q, $[]P$ signifie que la proposition P est toujours vraie le long de la séquence d'exécution, $\Diamond P$ signifie que la proposition P devient inévitablement vraie dans un état de la séquence d'exécution et $P \Rightarrow Q$ signifie

que si la proposition P est vraie alors la proposition Q est vraie. Après avoir donné les significations des combinateurs de la logique LTL utilisés, la formule LTL formalisant notre propriété est exprimée de la manière suivante :

$$[]\,(\,\textit{AttaqueEffectuée} \Rightarrow \Diamond\, \textit{AttaqueDétectée}\,)$$

Selon sa forme, cette propriété est considérée comme une propriété de *vivacité*. Une propriété de vivacité énonce que, sous certaines conditions, une formule finira par avoir lieu.

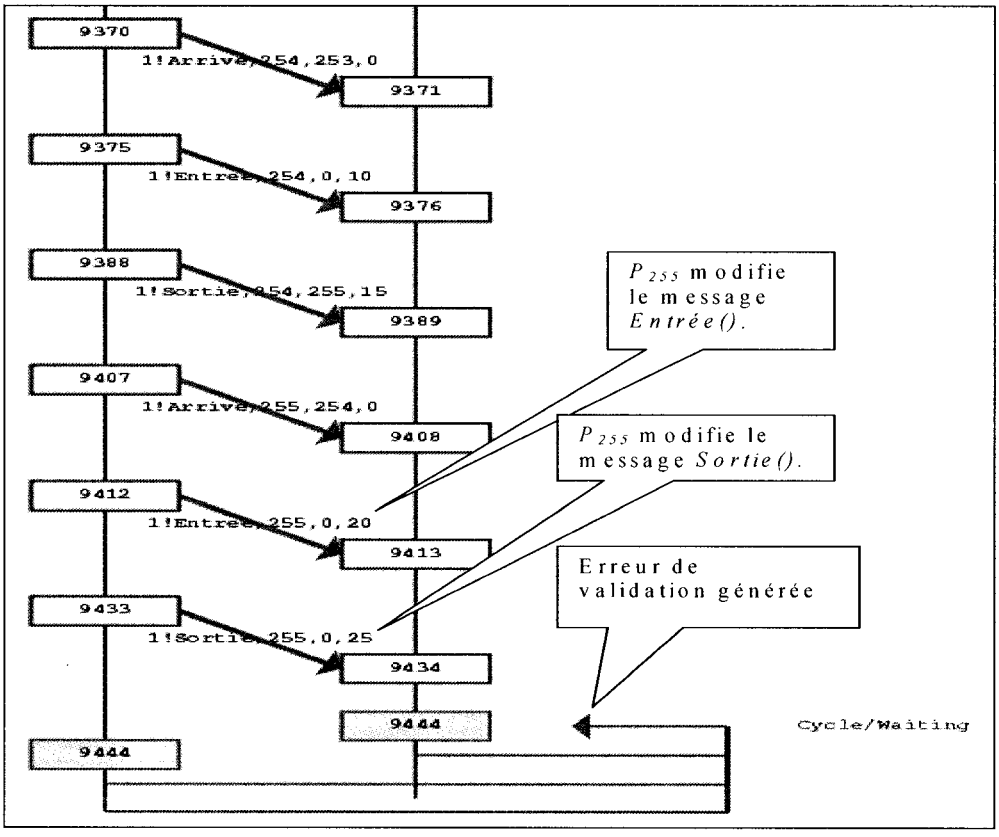

Figure 4.5 Explication de l'erreur de validation de la quatrième propriété, premier cas

Afin de vérifier la propriété pour chaque type d'attaque, nous activons une attaque à la fois. La validation a montré que la propriété est vérifiée pour toutes les attaques sauf dans deux cas où la plate-forme attaquante modifie les deux messages *Entrée()* et *Sortie()* ou bien elle modifie l'itinéraire de l'agent mobile en changeant la valeur de $Suiv_i$. La Figure 4.5 présente l'explication de l'erreur de validation générée dans le premier cas. La plate-forme P_{255} remplace la valeur de la donnée ($X=10$) par la valeur ($X'=20$) dans le message de type *Entrée()* et remplace la valeur de résultat ($R=15$) par la valeur ($R'=25$) dans le message de type *Sortie()*. L'agent sédentaire compare son résultat avec celui obtenu par l'agent mobile, réalise que les deux sont concordants et se trompe sur la fiabilité de la plate-forme P_{255}. Ce comportement menant à cette erreur de validation est prédit dans le scénario d'attaque 1-c détaillé dans le chapitre III.

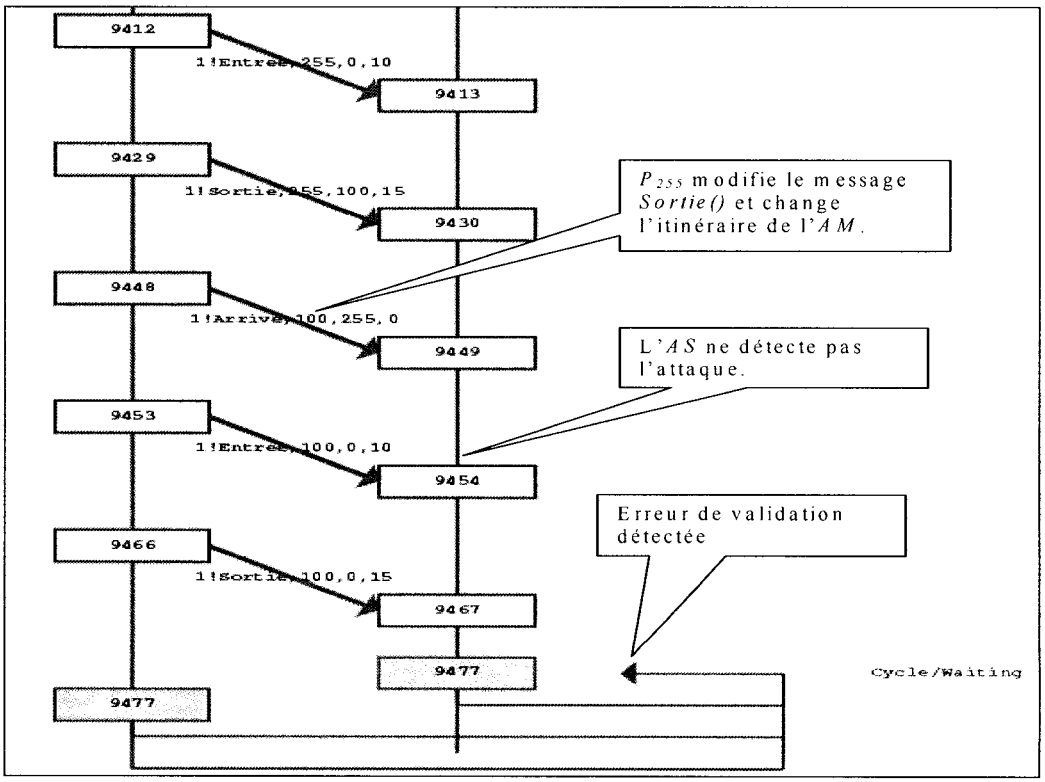

Figure 4.6 Explication de l'erreur de validation de la quatrième propriété, deuxième cas

La Figure 4.6 illustre l'explication de l'erreur de validation générée dans le deuxième cas. La plate-forme P_{255} modifie l'itinéraire de l'agent mobile en remplaçant l'identité de la plate-forme que l'agent mobile a choisie par l'identité d'une autre plate-forme. L'agent sédentaire *AS* ne détecte pas ce scénario d'attaque. Ce comportement concorde avec le scénario 8-c expliqué dans le chapitre III.

5. L'AS ne déclare jamais une détection tant qu'il n'y a pas une attaque

Avec cette dernière propriété, nous voulons vérifier que l'agent sédentaire *AS* ne déclare jamais une détection tant qu'il n'y a pas eu une attaque effectuée par une plate-forme malveillante. Nous avons formalisé cette propriété avec la logique temporelle linéaire LTL. En plus des deux propositions atomiques ajoutées pour valider la quatrième propriété, nous avons ajouté une autre proposition atomique *Fin* à notre modèle pour marquer la fin du protocole. Nous avons utilisé les combinateurs de la logique LTL : *toujours* [], la *négation* !, le *until* U et le *ou* V. Soient *P* et *Q* deux propositions. *PUQ* (*P until Q*) signifie que, le long de l'exécution courante, nous trouverons un état vérifiant *Q* et que tous les états rencontrés en attendant vérifieront *P*. La formule LTL formalisant notre propriété est exprimée de la manière suivante :

[] (! *AttaqueDétectée* U (*AttaqueEffectuée* V *Fin*))

Selon sa forme, cette propriété combine les aspects de sûreté et les aspects de vivacité. Une propriété de sûreté énonce que, sous certaines conditions, une formule ne se produit jamais.

Afin de vérifier la propriété pour chaque type d'attaque, nous activons une attaque à la fois. La validation a montré que la propriété est vérifiée pour toutes les attaques sauf dans le cas de l'attaque *déni de service* où la validation génère une erreur. La Figure 4.7 illustre l'explication de cette erreur de validation. Nous rappelons que l'agent sédentaire détecte ce type d'attaque à l'aide de compteur de temps *timeout*. Si ce

compteur de temps est initialisé avec une petite valeur, l'agent sédentaire *AS* détectera l'attaque *déni de service* même si la plate-forme courante P_1 n'effectue aucune attaque.

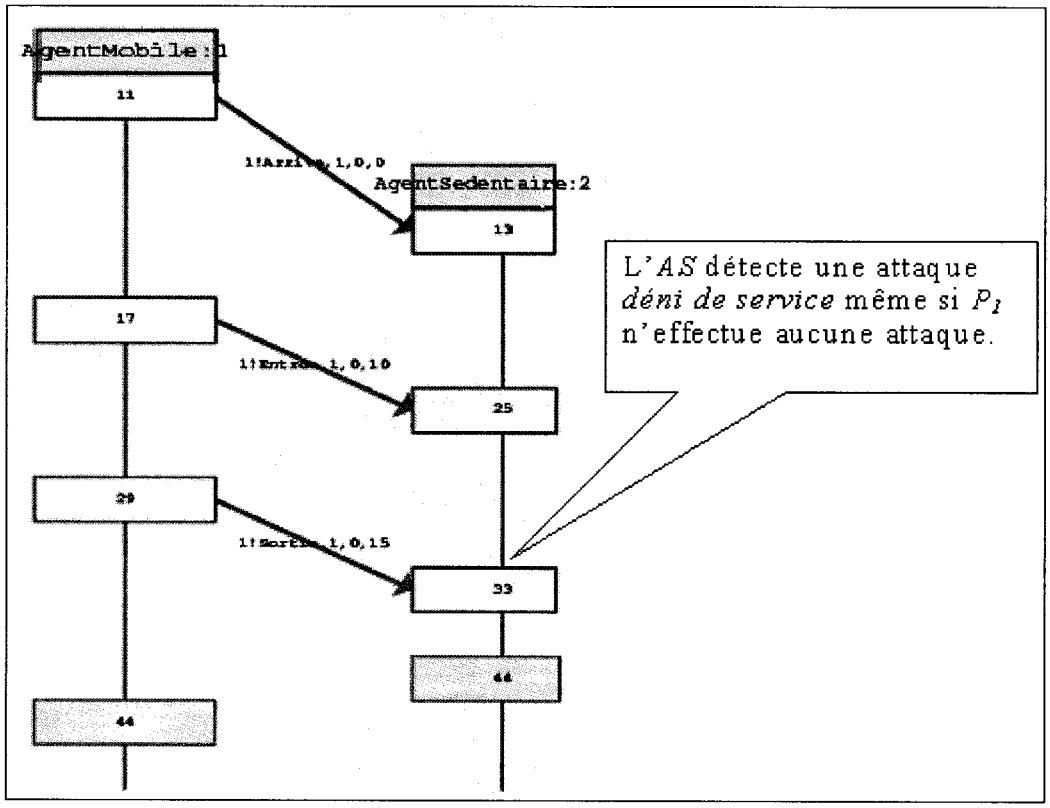

Figure 4.7 Explication de l'erreur de validation de la cinquième propriété

4.2 Implémentation et choix de mise en œuvre

Après avoir validé formellement notre protocole, nous l'avons implémenté pour s'assurer de ses fonctionnalités et pouvoir mesurer ses coûts en terme de trafic généré et de temps d'exécution. Nous avons pris comme exemple d'application un agent mobile qui se déplace dans plusieurs plates-formes et effectue un calcul critique à chaque exécution à l'intérieur des plates-formes visitées. Le choix d'une telle application est justifié, d'une part, par le fait qu'aucune application pratique nécessitant la sécurité du code et des données n'est disponible à notre niveau. D'autre part, cet exemple d'application a pour but de montrer la faisabilité de l'implémentation de notre protocole,

de tester la protection de l'agent mobile contre les attaques des plates-formes malveillantes et de mesurer l'efficacité et les coûts de nos algorithmes.

4.2.1 Environnement de développement

Nous avons utilisé la plate-forme d'agents mobiles *Grasshopper* pour implémenter notre protocole. Elle a été développée par la société allemande IKV, la première version a été disponible en août 1998. Son utilisation est gratuite pour des fins de recherche scientifique. Le langage de développement de la plate-forme est le langage Java, particulièrement en raison de sa portabilité. La programmation des agents mobiles se fait donc aussi en Java. Grasshopper est conforme au standard de l'OMG (Object Management Group) sur les agents mobiles. Cette norme a été conçue pour assurer l'interopérabilité entre les différentes plates-formes d'agents mobiles.

Grasshopper est un environnement d'agents répartis. Il est composé de régions, de places, d'agences et de deux types d'agents. Un agent peut être mobile ou sédentaire. L'agent mobile a l'habilité de se déplacer entre plusieurs plates-formes, quant à l'agent sédentaire, il n'a pas cette habilité. L'agence présente l'environnement d'exécution des agents mobiles et sédentaires. Une machine qui veut supporter les agents doit exécuter au moins une agence. Celle-ci peut contenir plusieurs places. Une place est un groupement logique des fonctionnalités à l'intérieur de l'agence. Le concept de la région facilite la communication entre les différents composants répartis dans Grasshopper, i.e. les agences, les places et les agents. Les agences et leurs places peuvent être associées à une région en les enregistrant dans son registre. Celui-ci enregistre automatiquement les créations, les suppressions et les déplacements des agents et des places appartenant à une agence associée à la région. La Figure 4.8 illustre la structure hiérarchique des composants de Grasshopper.

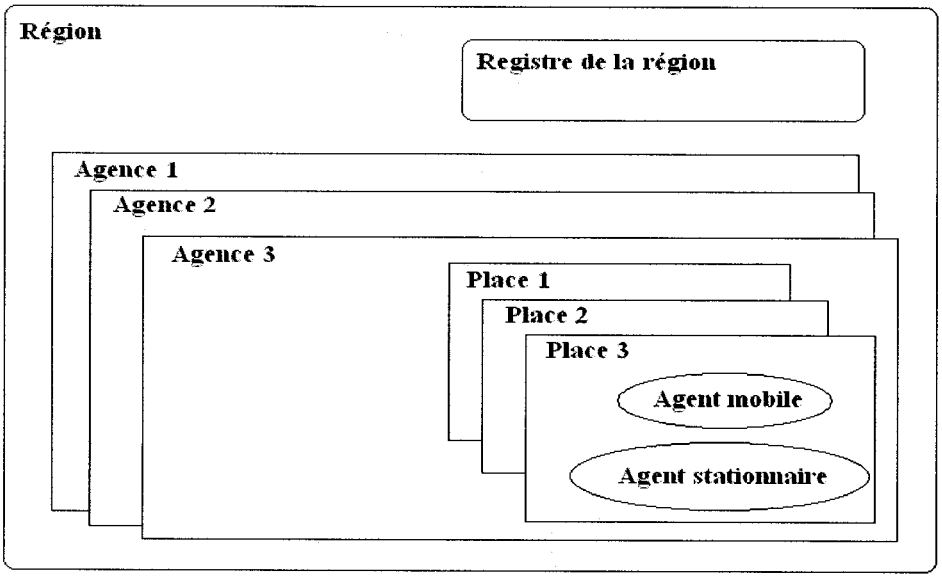

Figure 4.8 Structure hiérarchique des composants de Grasshopper

4.2.2 Communication entre les deux agents *AM* et *AS*

Suivant les spécifications du protocole décrites au chapitre précédent, l'agent mobile *AM* doit envoyer les trois messages *Arrive()*, *Entrée()* et *Sortie()* à son agent sédentaire coopérant *AS*. Pour implémenter ces envois, nous avons utilisé le service de communication, offert par la plate-forme Grasshopper, qui permet à un agent d'invoquer les méthodes d'un autre agent distant d'une manière transparente. Ce service de communication propose plusieurs protocoles de communications :

- CORBA IIOP (Common Object Request Broker Architecture – Internet Inter ORB Protocol) : il utilise un mécanisme conforme aux standards pour se connecter à un objet en utilisant le service de nom CORBA.
- MAF IIOP : ce protocole est une spécialisation de CORBA IIOP développée pour les interactions dans le cadre d'un système d'agents mobiles. Il a été décrit dans les standards MASIF et assure la connectivité entre des systèmes d'agents développés par des entreprises différentes.

- RMI (Remote Method Invocation) : il permet à des objets Java d'invoquer des méthodes d'autres objets, s'exécutant sur une autre machine virtuelle Java.
- RMI au-dessus de SSL : En utilisant ce protocole, RMI est exécuté au-dessus des sockets protégées par le protocole sécuritaire SSL. Celui-ci garantit le transport des données d'une manière cryptée.
- Socket : Ce type de communication est la plus robuste et la plus rapide car elle évite le surcoût d'un modèle d'objet réparti.
- Socket au-dessus de SSL : Les données transportées sont protégées par le protocole SSL.

Le *SSL* (*Secure Socket Layer*) [FRE96] est un protocole qui assure la confidentialité des données communiquées à travers le réseau Internet. Il est maintenant devenu un standard pour les communications où il y a un besoin de sécurité sur Internet. Il utilise les deux modes cryptographiques symétrique et asymétrique pour assurer la confidentialité, l'intégrité et l'authentification. Le transfert des données entre les deux entités communicantes est crypté avec une clé symétrique et un algorithme qui a été négocié lors de l'ouverture de session (*handshake*). Cela permet d'assurer la confidentialité. L'intégrité est garantie par l'utilisation de code d'authentification de messages qui prouve qu'un message n'a pas été modifié durant le transport. Le but de l'authentification est d'assurer que chaque paire d'entités communicantes soit mutuellement convaincue de l'identité l'une de l'autre. Pendant l'ouverture d'une session SSL, les deux entités échangent leurs identités et leurs clés publiques par l'intermédiaire d'un certificat qui suit la norme X.509 v3. Les identités sont acceptées lorsque le certificat a été signé par une autorité de certification de confiance.

Nous avons opté pour l'utilisation des simples *sockets* dans la communication entre les deux agents *AM* et *AS*, vu que c'est le moyen le plus rapide des protocoles offerts par la plate-forme Grasshopper. Nous avons évité d'utiliser le protocole sécuritaire SSL pour alléger la communication entre les deux agents. En effet, le protocole SSL augmente, d'une part, le temps d'établissement de la connexion entre les deux agents en échangeant les messages d'authentification, ainsi que le temps d'envoi et

de réception des données en encryptant et décryptant toutes les données de la session. D'autre part, le protocole encrypte toutes les données transportées dans la session alors que notre protocole n'a besoin d'encrypter qu'une partie des messages envoyés de l'agent mobile à son agent coopérant.

Cependant, pour assurer la confidentialité, l'intégrité et l'authentification des différents messages échangés, nous avons pris en charge ces services en les intégrant dans la conception de notre protocole et nous les avons implémentés suivant le besoin du protocole. De ce fait, nous ne cryptons que les données qui sont sensibles et dont l'espionnage portera atteinte au mécanisme de sécurité du protocole, par exemple les données fournies par une plate-forme, et qui sont requises à l'exécution de la partie critique du code de l'agent mobile, sont chiffrées avant de les envoyer. L'authentification et l'intégrité des données sont assurées en posant des signatures électroniques sur des éléments envoyés dans les trois types de messages. La signature électronique est calculée en appliquant une fonction de hashage sur l'élément concerné et le résultat obtenu est chiffré à l'aide de la clé privé de l'entité signataire.

4.2.3 Implémentation de la communication entre les deux agents

La communication des trois messages *Arrive()*, *Entrée()* et *Sortie()* entre les deux agents *AM* et *AS* passe à travers l'invocation distante des méthodes implémentées par l'agent coopérant *AS*. La classe de celui-ci, appelée *AgentStatio*, implémente l'interface *IAgentStatio* qui déclare les différentes méthodes invoquées par l'agent mobile afin d'envoyer les trois messages. La Figure 4.9 illustre l'interface *IAgentStatio*. Les méthodes *arrive()*, *entree()* et *sortie()* correspondent respectivement aux trois types de message *Arrive()*, *Entrée()* et *Sortie()* décrits lors de la spécification du protocole au chapitre III. La méthode *getRPA()*, qui retourne un objet de type *VectoreRPA* illustré à la Figure 4.9, permet à l'agent sédentaire *AS* d'envoyer les résultats sauvegardés et les identités des plates-formes attaquantes à l'agent mobile *AM* quand celui-ci termine son itinéraire et revient chez lui. La méthode *initTime()* permet d'initialiser les valeurs des compteurs temporels *timeout* et l'intervalle d'attente supplémentaire.

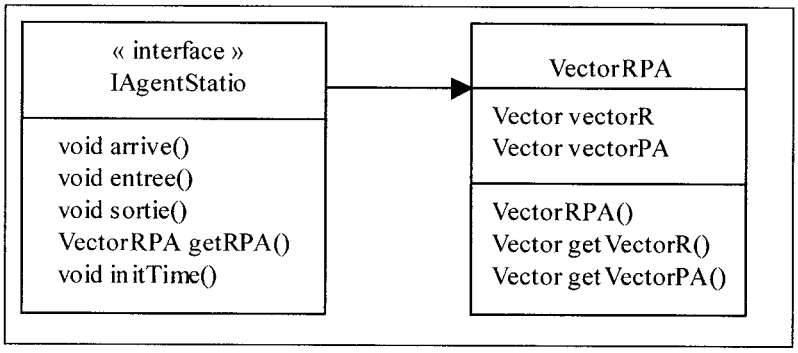

Figure 4.9 Présentation de l'interface *IAgentStatio* et la classe *VectorRPA*

L'envoi de ces messages utilise le service de communication de Grasshopper qui fonctionne de la manière illustrée à la Figure 4.10 : l'agent mobile *AM* ne peut interagir directement avec l'agent *AS* mais il construit un objet, dit « *proxy* », qui correspond à celui-ci. L'objet « *proxy* » a pour rôle d'établir la connexion avec l'agent sédentaire *AS* et rendre la communication transparente à l'agent mobile *AM*. Quand celui-ci veut envoyer un type de message, il invoque la méthode correspondante de « proxy » (Étape 1), qui va demander la localisation de l'agent sédentaire *AS* au service de domaine des agences (Étape 2), puis il va invoquer la méthode de l'agent *AS* à distance à travers le service de communication (Étape 3). Ce dernier va communiquer l'invocation par un protocole supporté par les deux agences (Étape 4). Puis, la méthode va être invoquée localement sur l'agent sédentaire *AS* (Étape 5).

La plate-forme Grasshopper offre les deux modes de communications synchrone et asynchrone. Dans une communication synchrone, quand un agent invoque une méthode d'un autre agent distant, il se bloque jusqu'à ce que ce dernier lui retourne le résultat. Par contre, lors de la communication asynchrone, l'agent n'attend pas à ce que l'agent distant lui envoie les résultats de l'invocation, mais il continue l'exécution de son code et il peut choisir le moment d'extraire ces résultats. Nous avons utilisé ce dernier mode pour communiquer les messages entre l'agent *AM* et l'agent *AS*. Ce mode a pour effet d'accélérer l'exécution de l'agent mobile qui n'est pas obligé d'attendre une réponse de son agent coopérant quand il s'exécute à l'intérieur d'une plate-forme visitée.

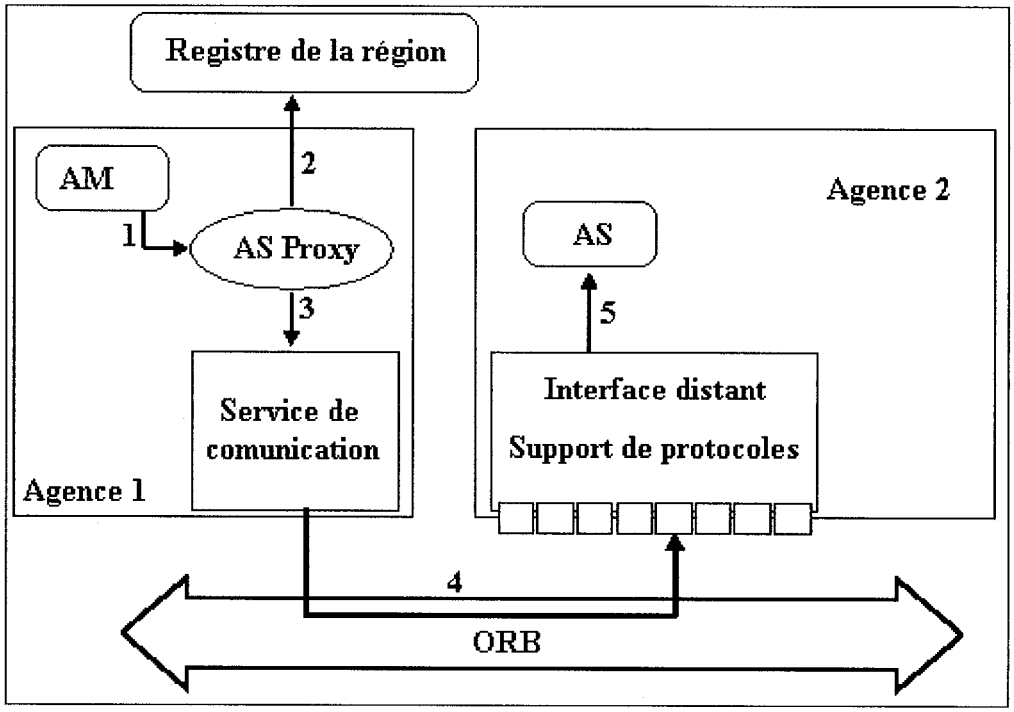

Figure 4.10 Communication entre *AM* et *AS* sous Grasshopper

4.2.4 Cryptage et signature électronique des messages échangés

Les messages envoyés par l'agent *AM* contiennent des données cryptées et d'autres signées par la plate-forme courante. De ce fait, l'agent mobile *AM* doit être en mesure de crypter et de signer les données sensibles et l'agent sédentaire *AS* doit avoir l'habilité de décrypter et de vérifier les signatures électroniques des messages reçus.

Nous avons utilisé les *API* de *JCE* (*Java Cryptography Extension*) [SCO01] fournies par SUN et celle développées par l'institut IAIK [IAI02]. Ces différentes librairies permettent le chiffrement et le déchiffrement des données, la signature électronique et la vérification de ces signatures ainsi que la gestion des clés et des certificats. Afin d'encrypter les données, nous avons opté pour l'algorithme *RSA* (Rivest, Shamir et Adleman) [PFL96] qui offre des fonctionnalités cryptographiques asymétriques. La signature électronique utilise l'algorithme *MD5withRSA*. Ces deux algorithmes sont considérés comme sécuritaires par la communauté scientifique au moment de la rédaction de ce mémoire. Nous avons généré une paire de clés de taille

égale à 1024 bits pour chaque plate-forme en utilisant l'utilitaire *Keytool* de SUN. Toutes les paires de clés et leur certificat sont sauvegardés dans un fichier appelé *keystore*.

Les deux agents *AM* et *AS* n'accèdent pas directement aux clés privées et publiques des plates-formes où ils sont entrain de s'exécuter mais font appel aux services fournis par une place particulière des agences de ces plates-formes. De ce fait, celles-ci ne sont pas obligées de divulguer leurs clés privées aux agents et préservent leur sécurité. La plate-forme Grasshopper offre la possibilité de créer des places avec des fonctionnalités particulières que seuls les agents qui sont sur cette place ont le pouvoir d'utiliser. Nous avons donc implémenté une place, appelée *CryptoPlace*, qui fournit les services de chiffrement et de signature à l'agent mobile *AM* et une place, appelée *CryptoPlaceTrust*, qui fournit les services de déchiffrement à l'agent sédentaire *AS*. La vérification des signatures est effectuée directement par l'agent *AS* vu qu'elle ne fait intervenir que la clé publique de la plate-forme signataire.

Les deux places utilisent la classe *KeyStoreHandler* pour charger le fichier *keystore* contenant les paires de clés, et par conséquent elles peuvent charger les clés publiques et privées des plates-formes. Nous avons utilisé la classe *SignedObject* pour signer un objet et vérifier cette signature. Cette classe fournie par JDK contient l'objet que nous voulons signer ainsi que sa signature. La Figure 4.11 illustre le diagramme des classes des deux places *CryptoPlace* et *CryptoPlaceTrust*.

4.2.5 Implémentation des classes des deux agents *AM* et *AS*

L'architecture implémentée comporte un agent mobile *AM* sécuritaire qui se déplace dans plusieurs plates-formes. C'est une classe, appelée *AMSecure*, qui hérite de la classe *MobileAgent* de Grasshopper pour pouvoir utiliser les fonctionnalités offertes par cette plate-forme. Sa méthode *init()*, que nous avons redéfinie, est exécutée une seule fois au moment de l'instanciation de la classe *AMSecure*. Elle cherche l'identifiant de l'agent sédentaire coopérant dans la région où l'agent mobile est enregistré. Puis, elle instancie un objet « *proxy* » représentant l'agent sédentaire distant en utilisant la classe

ProxyGeneratore afin de pouvoir faire appel aux méthodes de l'agent coopérant. Pour envoyer le message de type *Arrive()*, l'agent mobile utilise la méthode *afterMove()* que nous avons redéfinie. Cette méthode est invoquée automatiquement par l'agence recevant l'agent mobile après une procédure de migration réussie.

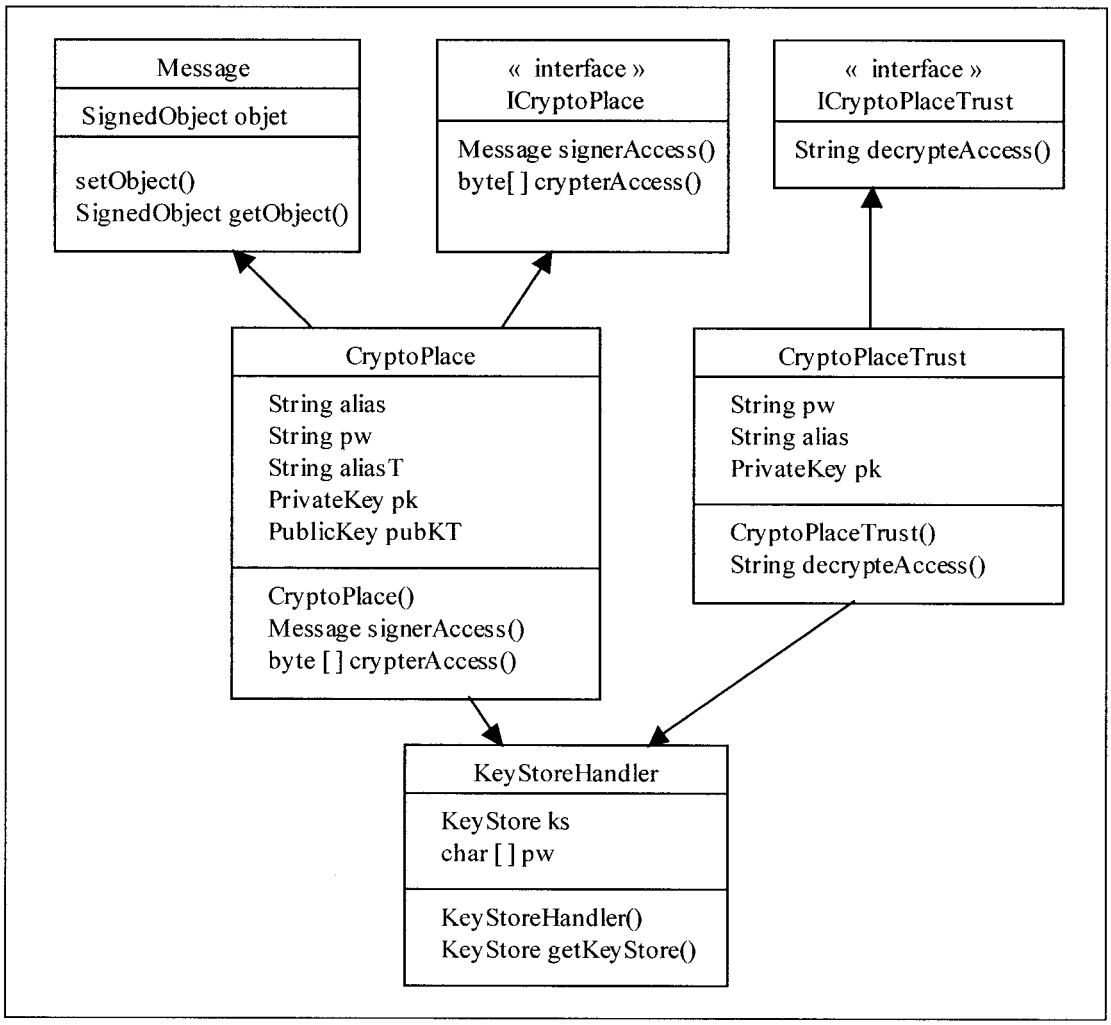

Figure 4.11 Diagramme de classe *CryptoPlace* et *CryptoPlaceTrust*

À chaque étape d'exécution à l'intérieur d'une plate-forme, l'agent mobile invoque les méthodes *signerAccess()* et *crypterAccess()* implémentées par la place *CryptoPlace* pour crypter les données et calculer la signature électronique. Puis, elle

invoque les méthodes distantes *entree()* et *sortie()* pour envoyer les messages de type *Entrée()* et *Sortie()*. La Figure 4.12 présente le diagramme des classes de l'agent *AM*.

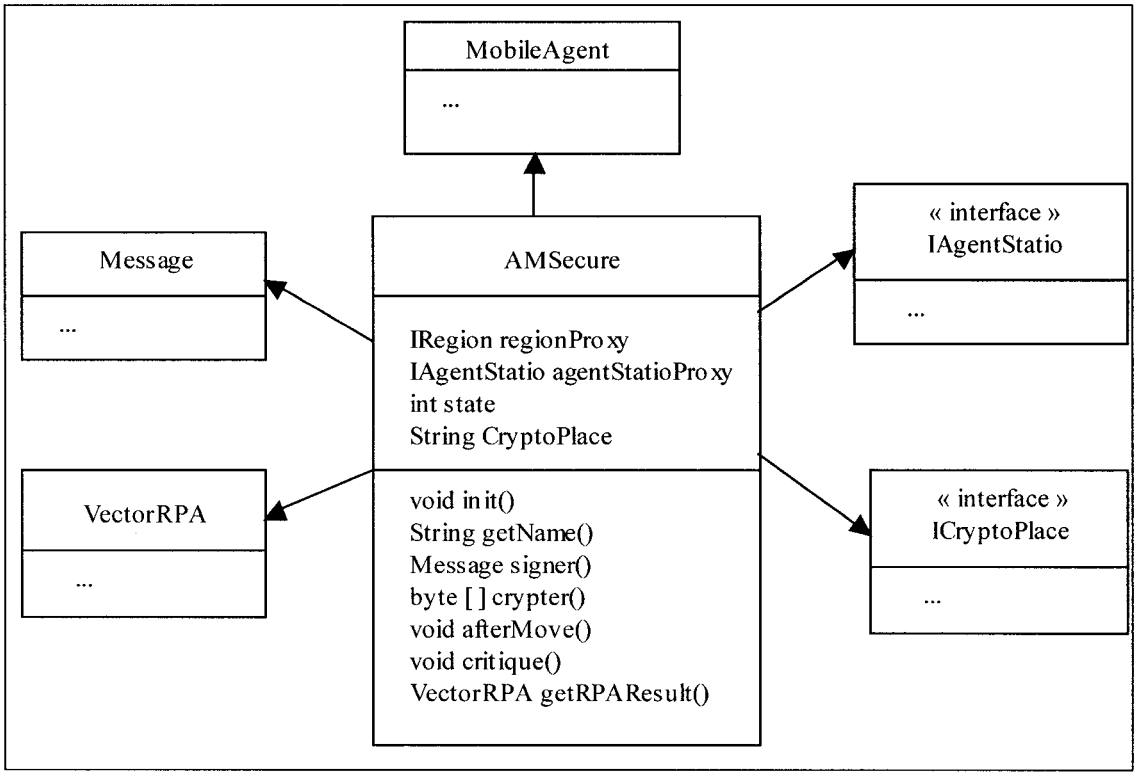

Figure 4.12 Diagramme de classe de l'agent *AM*

La classe *AgentStatio* implémente les fonctionnalités de l'agent sédentaire coopérant *AS*. Elle hérite de la classe *MobileAgent* de Grasshopper pour pouvoir effectuer son seul déplacement de la plate-forme d'origine *O* vers la tierce plate-forme fiable *T*. Elle implémente toutes les méthodes déclarées dans l'interface *IAgentStatio,* et qui permettent à l'agent mobile de lui envoyer les différents messages. La méthode *codeCritique()* est une duplication du code critique de l'agent mobile, l'agent sédentaire *AS* s'en sert pour vérifier les résultats obtenus par l'agent *AM*. Toutes les vérifications effectuées par l'agent sédentaire *AS* concernant la détection des attaques sont implémentées dans les méthodes *arrive()*, *entrée()* et *sortie()*. Ces vérifications sont donc effectuées au fur et à mesure que l'agent *AS* reçoit les données envoyées par son agent

mobile *AM*. Afin de décrypter les données reçues, l'agent *AS* fait appel aux services cryptographiques offerts par la place *CryptoPlaceTrust* décrite dans les sections précédentes. L'agent *AS* vérifie directement les signatures électroniques des messages en chargeant la clé publique de la plate-forme signataire. La Figure 4.13 présente le diagramme de classe de l'agent sédentaire coopérant *AS*.

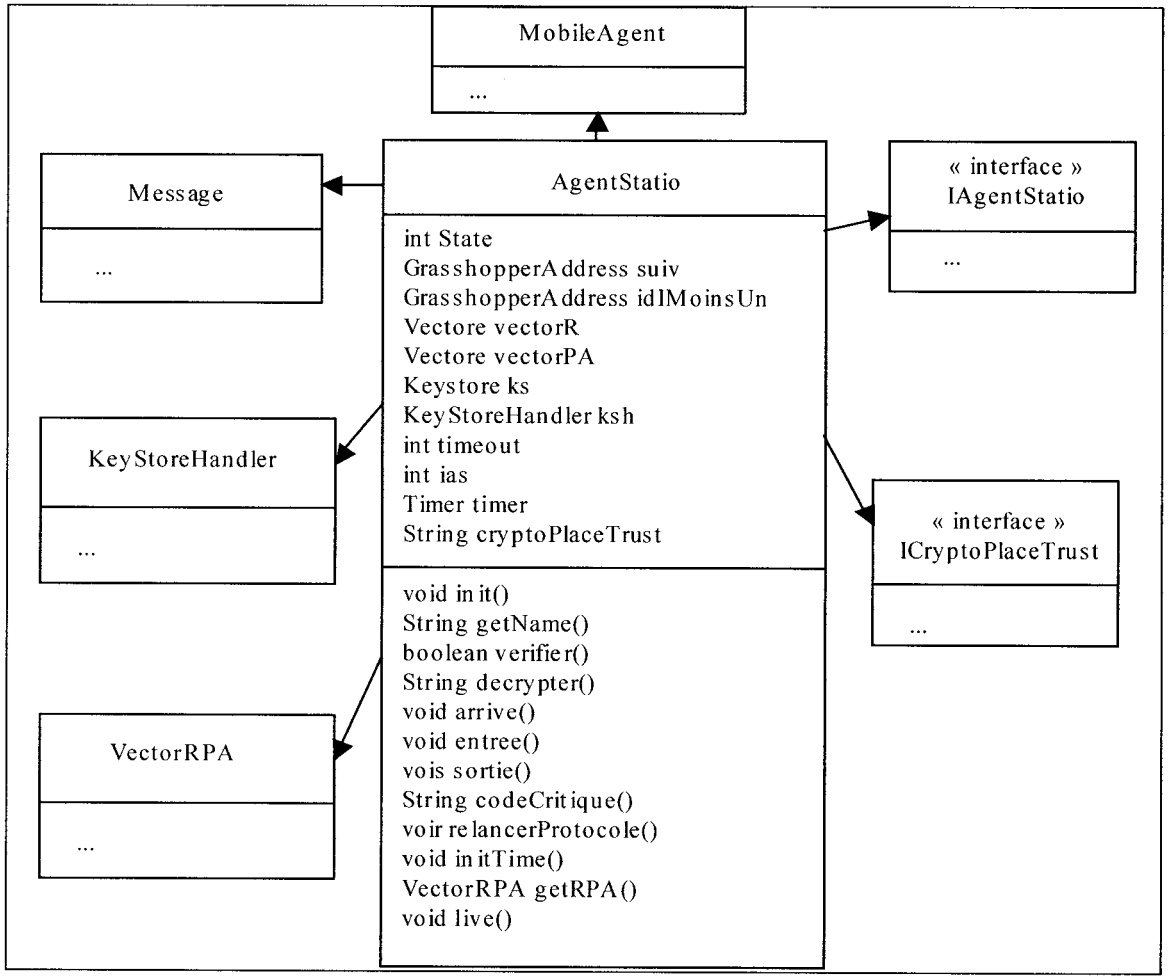

Figure 4.13 Diagramme de classe de l'agent *AS*

4.2.6 Implémentation et estimation des compteurs temporels *timeout* et l'*IAS*

Il est important de rappeler que le protocole utilise les compteurs temporels

timeout et l'intervalle d'attente supplémentaire (IAS) pour détecter les attaques de la ré-exécution du code de l'agent mobile et de déni de service. L'agent sédentaire *AS* déclenche le compteur *timeout* après la réception du message *Arrive()* afin de limiter le temps d'exécution de l'agent mobile *AM* à l'intérieur de la plate-forme courante. Nous avons utilisé la classe *Timer* [HOL00] fournie dans le paquetage *Swing* de SUN pour implémenter ces compteurs temporels. Cette classe permet de notifier le *thread* qui exécute l'agent *AS* dans un intervalle de temps défini préalablement. Au moment de la déclaration, elle est initialisée avec une valeur de temps en milli-seconde représentant la valeur initiale du compteur *timeout*. À la réception du message *Arrive()*, l'agent *AS* invoque la méthode *start()* de l'objet *Timer* pour déclencher le compteur. Si le temps calculé à partir du déclenchement du compteur dépasse la valeur initiale de délai, l'objet *Timer* exécute sa méthode *actionPerformed()* qui permet à l'agent *AS* de détecter l'attaque de la ré-exécution du code et en même temps déclencher le compteur d'intervalle d'attente supplémentaire (IAS). Après l'expiration de ce dernier intervalle, l'objet *Timer* notifie l'agent *AS* pour qu'il puisse détecter l'attaque déni de service et par conséquent relancer le protocole.

Pour pouvoir utiliser les compteurs temporels, il fallait calculer, ou au moins estimer, la valeur de *timeout* et de l'*IAS*. Nous rappelons que le *timeout* présente le temps maximum nécessaire pour exécuter le code de l'agent mobile et transmettre les deux messages *Entrée()* et *Sortie()*, et que l'*IAS* présente l'intervalle de temps après lequel l'agent sédentaire *AS* détecte l'attaque déni de service. Nous avons implémenté un agent, appelé *agent estimateur* (*AE*), qui permet à la plate-forme d'origine *O* d'estimer la valeur de *timeout*. Nous n'avons pas estimé la valeur de l'*IAS* vu que celle-ci dépend de l'application qui implémentera le protocole. Il reste à la guise du concepteur de déterminer le délai à partir duquel il considère le temps d'attente comme une attaque déni de service.

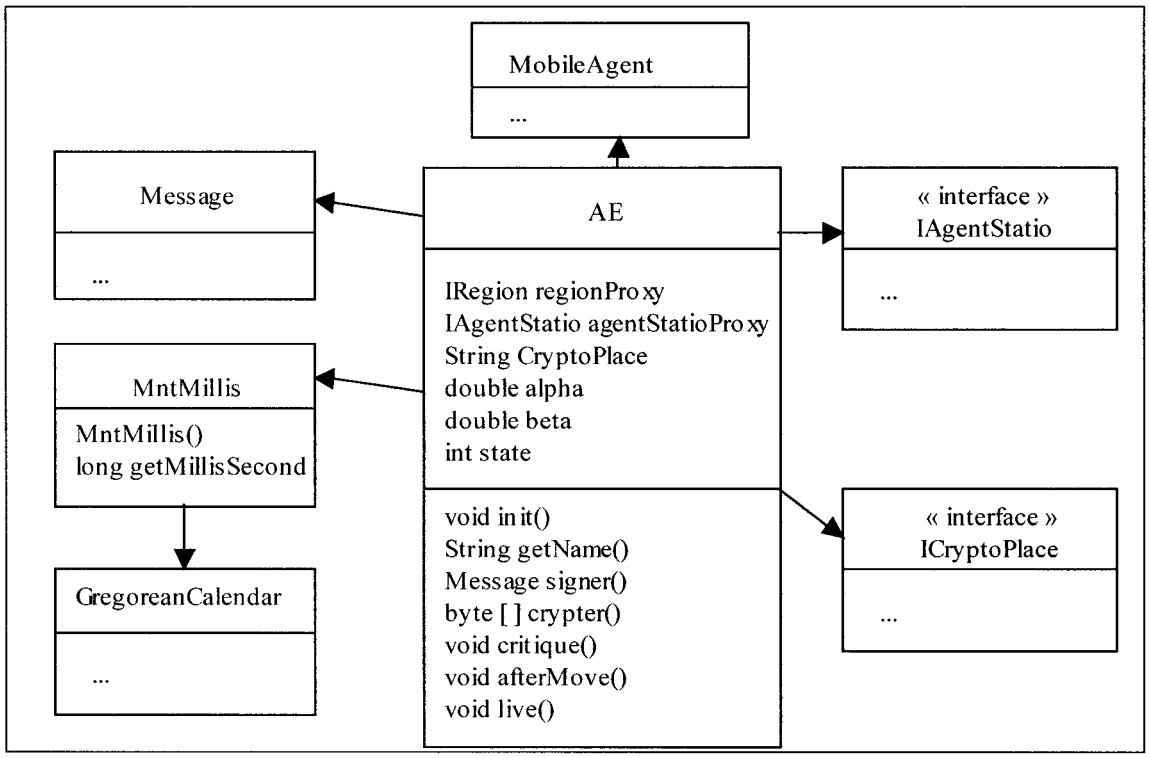

Figure 4.14 Diagramme de classe de l'agent *AE*

Au début du protocole, la plate-forme d'origine *O* envoie l'agent sédentaire *AS* vers la tierce plate-forme fiable *T*. Puis, elle exécute l'agent *AE* qui se charge d'estimer la valeur *timeout* et communique son résultat à l'agent *AS*. Afin de calculer cette estimation, l'agent *AE* simule une étape d'exécution de l'agent mobile *AM* à l'intérieur d'une plate-forme. Il crée l'objet « *proxy* » pour qu'il puisse communiquer avec l'agent *AS*, utilise les services cryptographiques de la place *CryptoPO* créée dans la plate-forme d'origine *O* pour encrypter les données et calculer les signatures électroniques, exécute la partie critique de l'agent mobile et envoie les deux types de messages *Entrée()* et *Sortie()*. L'agent *AE* prélève le temps système au début de son exécution par la méthode *getMillisSecond()* définie dans la classe *MntMillis* qui hérite de la classe *GregorianCalendar*. À la fin de l'exécution de la partie critique, il prélève le temps système une deuxième fois pour qu'il puisse estimer le temps d'exécution de l'agent

mobile depuis son arrivée à une plate-forme jusqu'à la fin d'exécution de sa partie critique. La Figure 4.14 illustre le diagramme de classes de l'agent *AE*.

Nous avons mené des expériences concernant la détection de la ré-exécution du code de l'agent mobile en affectant directement la valeur estimée par l'agent *AE* au compteur temporel *timeout*. Les résultats ont révélé que cette simple affectation ne donne pas des taux satisfaisants. Les taux étudiés sont le pourcentage où l'agent *AS* détecte l'attaque de la ré-exécution et le pourcentage où il marque une plate-forme attaquante alors que celle-ci n'a exécuté le code qu'une seule fois. Pour étudier ce phénomène, nous avons effectué l'expérience suivante :

> L'agent mobile *AM* se déplace dans deux plates-formes P_1 et P_2. Nous avons forcé la plate-forme P_1 de ré-exécuter le code critique de l'agent mobile alors que la plate-forme P_2 l'exécute une seule fois. Nous avons calculé le temps d'exécution de l'agent estimateur *AE* ainsi que les deux temps d'exécution de l'agent *AM* à l'intérieur des deux plates-formes P_1 et P_2.

La Figure 4.15 présente les résultats obtenus.

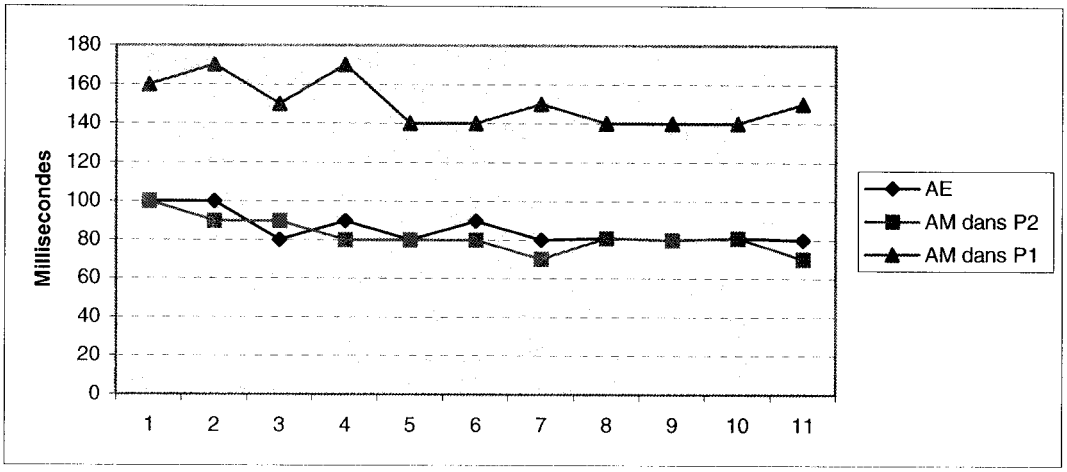

Figure 4.15 Temps d'exécution des agents *AM* et *AE*

Nous remarquons que les deux courbes représentant le temps d'exécution de l'agent *AE* et le temps d'exécution de l'agent *AM* sur la plate-forme P_2 sont presque

confondus. Ceci est dû au fait que les deux agents exécutent le même code dans des environnements pareils. Le chevauchement des deux courbes justifie les taux insatisfaisants des premières expériences. Par contre, la courbe représentant le temps d'exécution de l'agent *AM* sur la plate-forme attaquante P_1 est nettement éloignée des deux courbes. Ceci est justifié par le fait que la plate-forme P_1 ré-exécute le code et donc elle passe plus de temps.

Suite aux résultats de l'expérience précédente, nous avons décidé d'affecter à la valeur de *timeout* la multiplication d'un coefficient avec la valeur estimée par l'agent *AE* (***timeout* = coefficient × temps d'exécution de *AE***). Nous avons varié ce coefficient de 0.8 jusqu'à 2 pour étudier l'impact de ce coefficient sur les deux taux, i.e. le taux de la détection de la ré-exécution et celui de la fausse détection de la ré-exécution. Les résultats sont présentés à la Figure 4.16.

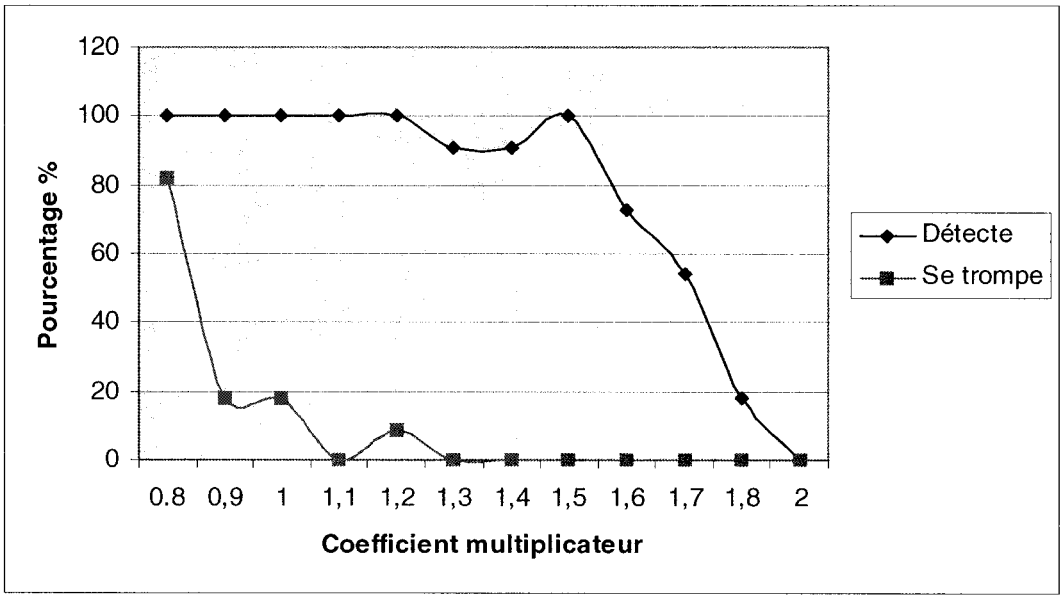

Figure 4.16 Impact du coefficient multiplicateur sur les deux taux

Nous avons remarqué que, lorsque la valeur du coefficient est faible, le taux de détection de la ré-exécution est égal à 100% mais le taux où l'agent *AS* se trompe est élevé (entre 20% et 80%). Ceci est expliqué par le fait que le compteur *timeout* qui a une valeur petite dans ce cas-ci expire avant que l'agent *AM* termine son exécution à

l'intérieur des deux plates-formes P_1 et P_2. Par contre, quand le coefficient est grand, les deux taux deviennent nuls. L'agent sédentaire coopérant *AS* ne se trompe pas sur la fiabilité de la plate-forme P_2 mais il ne détecte plus la ré-exécution du code par P_1. Ceci est dû à la grande valeur de *timeout* qui n'expire qu'après la fin des exécutions de l'agent *AM*. Le meilleur intervalle est situé entre 1,3 et 1,5 où le taux de détection de l'attaque est supérieur à 90% et le taux où l'agent *AS* se trompe égale à 0%. Ce résultat est justifié par le fait que les environnements d'exécution varient entre les différentes plates-formes P_1, P_2 et *O*. Le temps d'exécution d'un agent dépend du nombre de processus actifs sur la plate-forme, du nombre des agents actifs, des caractéristiques techniques des machines et de l'état du réseau au moment des transferts de l'agent *AM* et des messages.

4.3 Tests et évaluation de l'implémentation

Pour faire nos tests, nous avons utilisé 4 machines dont les principales caractéristiques sont décrites au Tableau 4.1. La première machine exécute l'agence de la plate-forme d'origine *O*. La deuxième exécute l'agence de la tierce plate-forme fiable *T* et les deux autres exécutent les agences des deux plates-formes P_1 et P_2 constituant l'itinéraire de l'agent mobile *AM*. Le réseau utilisé est un réseau local de type Ethernet 100 Mbps.

Tableau 4.1 Description des machines utilisées pour les tests

Type	Nom de la plate-forme	RAM (Mo)	Processeur	Système d'exploitation
Ordinateur de bureau	Plate-forme d'origine O	256	P IV, 1.7 GHz	Windows 2000 pro
Ordinateur de bureau	Tierce plate-forme fiable T	256	P IV, 1.7 GHz	Windows 2000 pro
Ordinateur de bureau	Plate-forme P1	256	P IV, 1.7 GHz	Windows 2000 pro
Ordinateur de bureau	Plate-forme P2	256	P IV, 1.7 GHz	Windows 2000 pro

4.3.1 Tests de l'implémentation

Nous avons testé les fonctionnalités du protocole liées à la détection des différentes attaques d'une plate-forme malveillante. Pour se faire, nous avons élaboré un jeu de tests qui se base sur les scénarios d'attaque que nous avons pu imaginer lors de la phase de la spécification du protocole. Dans chaque scénario, nous forçons la plate-forme P_1 ou P_2 à attaquer l'agent mobile de la même manière que la description de ce scénario. Le Tableau 4.2 présente le jeu de tests et les résultats obtenus.

Tableau 4.2 Jeu de tests et résultats

Scénario d'attaque	Implémentation	Résultat
Modification de message *Entrée()*	P_2 envoie $X'=4$ au lieu de $X=2$	L'agent *AS* détecte l'attaque
Modification de message *Sortie()*	Après le calcul des résultats, P_2 envoie une valeur aléatoire	L'agent *AS* détecte l'attaque
Modification de message *Entrée()* et *Sortie()*, et ré-exécution du code	P_1 ré-exécute le code	L'agent *AS* détecte l'attaque si la valeur de *timeout* est bien estimée
Exécution incorrecte du code	P_1 et P_2 exécutent un code modifié	L'agent *AS* détecte l'attaque
Changement des messages par un intrus	P_1 signe avec une clé privée qui n'est pas la sienne	L'agent *AS* détecte l'attaque
Déni de service	P_1 retarde l'exécution de l'agent *AM* en utilisant la méthode *sleep*	L'agent *AS* détecte l'attaque si la valeur de l'*IAS* est adéquate
Analyse et ré-exécution du code	P_1 ré-exécute le code	L'agent *AS* détecte l'attaque si la valeur de *timeout* est bien estimée
Modification d'itinéraire sans mascarade et sans modification de $Suiv_i$	P_1 envoie l'agent *AM* à P_3 au lieu de P_2 et $Suiv_1 = P_2$	L'agent *AS* détecte l'attaque
Modification d'itinéraire avec mascarade et sans modification de $Suiv_i$	P_1 envoie l'agent *AM* à P_3 au lieu de P_2 et $Suiv_1 = P_2$. P_3 envoie l'identité de P_2 au lieu de la sienne	L'agent *AS* détecte l'attaque
Modification d'itinéraire avec modification de $Suiv_i$	P_1 envoie l'agent *AM* à P_3 au lieu de P_2 et $Suiv_1 = P_3$.	L'agent *AS* ne détecte pas l'attaque

En analysant le Tableau 4.2, nous constatons que l'agent sédentaire coopérant *AS* détecte toutes les attaques dans les différents scénarios, sauf dans le cas de l'attaque de la modification de l'itinéraire avec modification de la valeur $Suiv_i$. La justification de ce dernier résultat est détaillée dans le scénario d'attaque 8-b (chapitre III). Il est à noter que l'agent *AS* ne parvient pas à détecter le dernier scénario dans le seul cas où l'itinéraire est dynamique. En effet, si l'itinéraire est statique, la plate-forme d'origine *O* pourra communiquer la liste des plates-formes constituant l'itinéraire de l'agent mobile *AM* à l'agent sédentaire *AS*, et par conséquent cette dernière attaque devient détectable.

4.3.2 Évaluation de l'implémentation

Nous avons implémenté notre protocole en prenant comme exemple d'application un agent mobile *AM* qui visite deux plates-formes P_1 et P_2, exécute un code critique à l'intérieur de chaque plate-forme et retourne chez lui. Les performances de l'implémentation de notre protocole sont comparées avec celles d'un agent mobile simple, appelé *AMSimple*, qui effectue le même parcourt que l'agent mobile *AM* mais qui n'est pas sécuritaire, i.e. n'implémente pas notre protocole. Nous avons pris comme métrique le trafic généré par l'agent mobile et le temps d'exécution de celui-ci. Le choix de ces deux métriques est justifié par le fait que notre protocole génère, d'une part, un trafic additionnel dû aux échanges de messages entre l'agent mobile et l'agent coopérant et, d'autre part, il ajoute un temps d'exécution dû aux traitements de la détection des attaques. Cet exemple d'application et ces mesures vont nous permettre d'étudier l'impact de l'ajout de notre protocole à une application pour la sécuriser.

Analyse du trafic généré

Nous avons utilisé le logiciel EtherPeek v4.5, développé par la société WildPackets.com, pour analyser le trafic généré par notre protocole. Ce logiciel permet de capturer tous les paquets qui transitent dans le réseau en fournissant des outils de filtrage sur les ports, les protocoles et les adresses source et destination. Il fournit aussi des statistiques et des graphiques représentant la répartition du trafic suivant certains

paramètres préalablement définis par l'utilisateur. Nous avons mesuré le nombre d'octets communiqués entre les différentes plates-formes (O, T, P_1 et P_2) en filtrant les paquets sur le port d'écoute des agences et sur les adresses de destinations. La Figure 4.17 présente les tailles moyennes des agents *AM*, *AS* et *AMSimple*. La taille moyenne d'un agent est égale à la somme des octets générés par l'agent mobile lors de ses déplacements divisée par le nombre de déplacements.

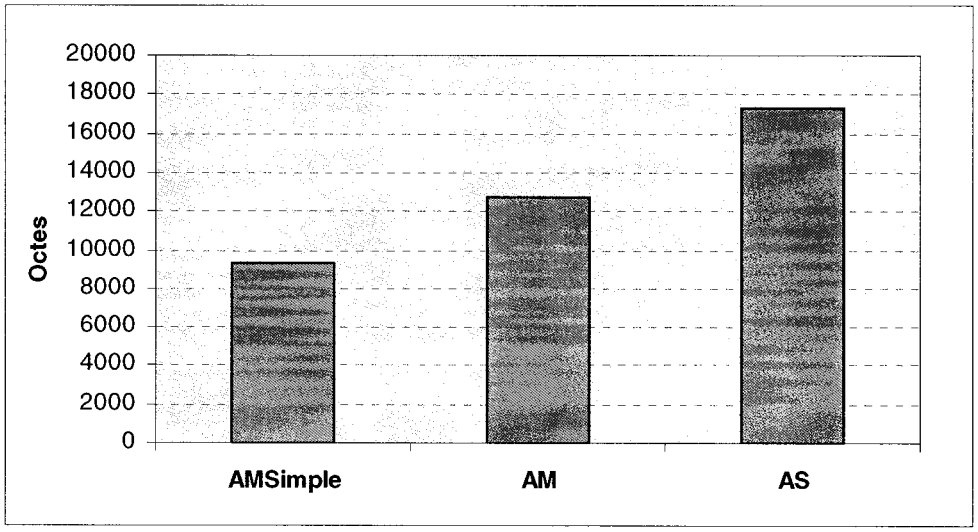

Figure 4.17 Comparaison des tailles des agents *AM*, *AS* et *AMSimple*

En analysant la Figure 4.17, nous constatons que la taille moyenne de notre agent sécuritaire *AM* (~12 ko) n'est pas très grande par rapport à la taille moyenne de l'agent simple *AMSimple* (~9 ko). Ceci est justifié par le fait que notre agent *AM* n'implémente qu'une petite partie de sa sécurité et la majorité des traitements est supportée par son agent sédentaire coopérant *AS*. En effet, les seuls ajouts dans le code de l'agent mobile *AM* sont les échanges de messages, et les fonctionnalités cryptographiques sont supportées par les plates-formes visitées elles-même. Le deuxième constat est la grande taille de l'agent sédentaire *AS* (~17 ko) justifiée par la même raison, i.e. l'agent *AS* effectue tout le traitement de la sécurité. Mais, même si la taille de l'agent sédentaire est grande, elle n'influence pas beaucoup le trafic total généré, vu que ce dernier agent ne se déplace qu'une seule fois entre la plate-forme d'origine O et la tierce plate-forme T.

Les mesures prises avec l'analyseur EtherPeek nous ont permis de connaître la répartition entre les différentes plates-formes (*O*, *T*, P_1 et P_2) du trafic généré par l'agent *AM*. La Figure 4.18 illustre cette répartition. Nous remarquons que le trafic entre la plate-forme d'origine *O* et la plate-forme fiable *T* constitue la plus grande partie du trafic (35%). Ceci est justifié, d'une part, par le déplacement de l'agent sédentaire *AS* dont la taille est grande, d'autre part par les échanges additionnels des messages par l'agent estimateur *AE*, et le dernier message contenant les résultats et les identités des plates-formes attaquantes envoyé par l'agent *AS*. La deuxième remarque est que le trafic généré par les échanges des messages *Arrive()*, *Entrée()* et *Sortie()* ne constitue que 9% du trafic total. Ceci est justifié par le nombre réduit des éléments composant ces messages. En effet, l'agent mobile *AM* envoie seulement les objets permettant à son agent coopérant de détecter les attaques.

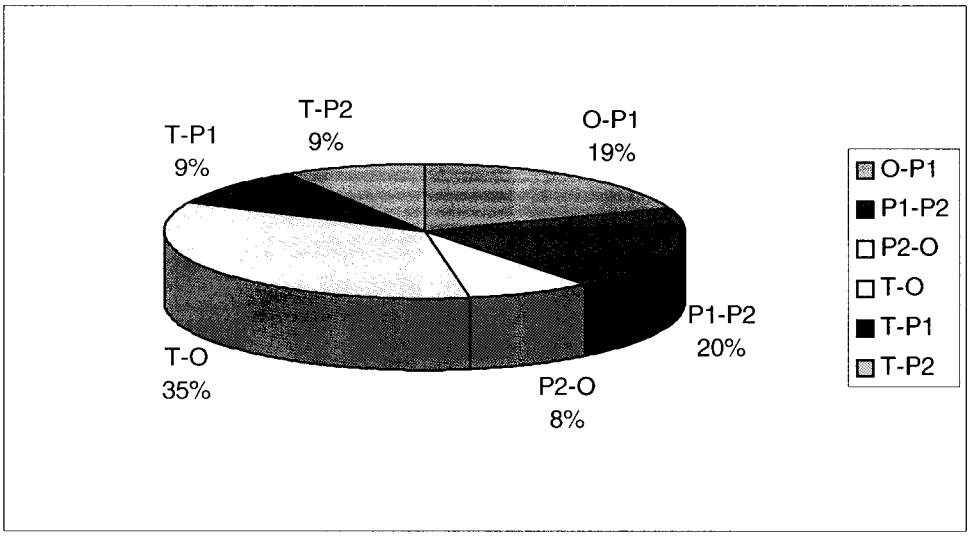

Figure 4.18 Répartition du trafic entre les différentes plates-formes

D'après ces résultats, nous concluons que l'ajout des fonctionnalités de la sécurité de notre protocole à une application d'agent mobile n'aura pas une grande influence sur le trafic généré par l'application elle-même. De plus, la sécurisation du code et des données de l'application tolérera le trafic additionnel.

Analyse du temps d'exécution

La deuxième métrique utilisée pour étudier les performances de l'implémentation de notre protocole est le temps d'exécution de l'agent mobile *AM*. Nous avons calculé le temps global d'exécution de l'agent sécuritaire *AM* qui correspond à la durée d'exécution de l'agent depuis sa création par la plate-forme d'origine *O* jusqu'à sa suppression par la même plate-forme. Nous avons ensuite comparé ces résultats avec le temps global d'exécution de l'agent simple *AMSimple*. Les deux agents prélèvent le temps système au début de leur exécution par la méthode *getMillisSecond()* définie dans la classe *MntMillis*. Ils font appel à la méthode *getMillisSecond()* à partir de leur méthode *init()*. À la fin de leur parcours, il prélève le temps système une deuxième fois avant leur suppression pour qu'il puisse calculer le temps global d'exécution. Dans cette dernière étape, ils font appel à la méthode *getMillisSecond()* à partir de leur méthode *beforeRemove()*.

Lorsque nous avons pris les mesures, nous avons constaté qu'il y avait de fortes variations du temps d'exécution entre la première mesure et celles qui suivaient. Ceci est dû à la machine virtuelle Java qui met en cache les classes qu'elle charge en mémoire. Lors de la prochaine instantiation de cette classe, la machine virtuelle ira chercher la classe dans sa cache et le temps pour instantier cette classe sera beaucoup plus rapide. Pour remédier à ce phénomène et accélérer le temps de réponse de l'agent mobile *AM* lors de sa première exécution, nous forçons la machine virtuelle Java de charger les classes de la sécurité requises à l'exécution de l'agent mobile pendant la création de l'agence. Lorsque nous créons l'agence, nous créons aussi la place cryptographique *CryptoPlace* et nous exécutons un agent qui fait appel aux classes de la sécurité. De cette manière, nous assurons l'homogénéité des mesures prises.

Nous avons comparé le temps global d'exécution de l'agent sécuritaire *AM* implémentant notre protocole avec celui de l'agent *AMSimple*. La Figure 4.19 illustre cette comparaison. Nous constatons que les deux temps d'exécution sont très proches, une différence de 250 milli-secondes constituant une augmentation de 5,8% du temps d'exécution de l'agent *AM* par rapport à l'agent *AMSimple*, qui représente un très bon

résultat. Ceci est expliqué par trois facteurs. Tout d'abord, l'agent mobile utilise une communication asynchrone pour envoyer les messages à son agent coopérant *AS*, ce qui accélère l'exécution de l'agent à l'intérieur des plates-formes. Ensuite, la manière dont nous avons conçu le protocole permet à l'agent *AM* d'exécuter son code sans attendre une réponse de la part de son agent coopérant, ce qui évite le blocage de notre agent mobile *AM*. Enfin, les mécanismes de migration et d'instanciation de l'agent mobile sont beaucoup plus coûteux en terme de temps de réponse que l'exécution de l'agent à l'intérieur des plates-formes. En effet, en mesurant le temps d'exécution de l'agent mobile *AM* à l'intérieur d'une plate-forme, nous avons constaté que ce temps ne représente que 1,8% du temps global d'exécution.

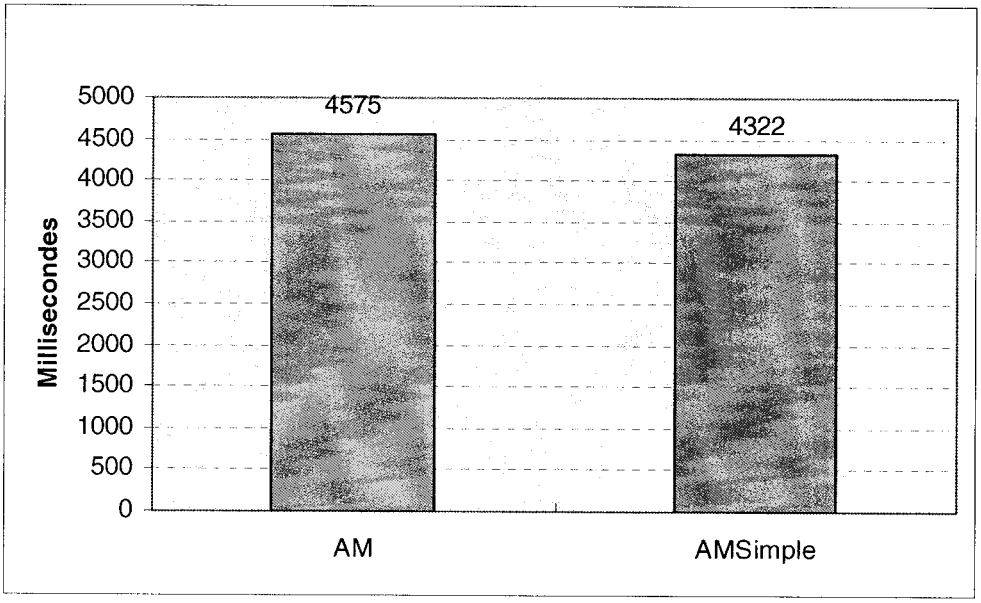

Figure 4.19 Comparaison des temps d'exécution des agents *AM* et *AMSimple*

Dans une dernière mesure, nous avons étudié l'impact du nombre des agents actifs au même moment dans les plates-formes visitées par l'agent mobile *AM* et dans la plate-forme d'origine *O*. Nous avons comparé ensuite les résultats obtenus avec ceux de l'agent *AMSimple*. La Figure 4.20 illustre cette étude. Nous constatons que les temps globaux d'exécution des deux agents augmentent avec l'accroissement du nombre des agents actifs, et les écarts entre les temps d'exécution des deux agents augmentent aussi.

Ceci est justifié par le fait que la plate-forme Grasshopper implémente les agents mobiles et sédentaires sous forme de « thread » Java avec les plus faibles priorités, qui désavantage les exécutions des agents par rapport à d'autres threads et processus qui ont une priorité plus grande.

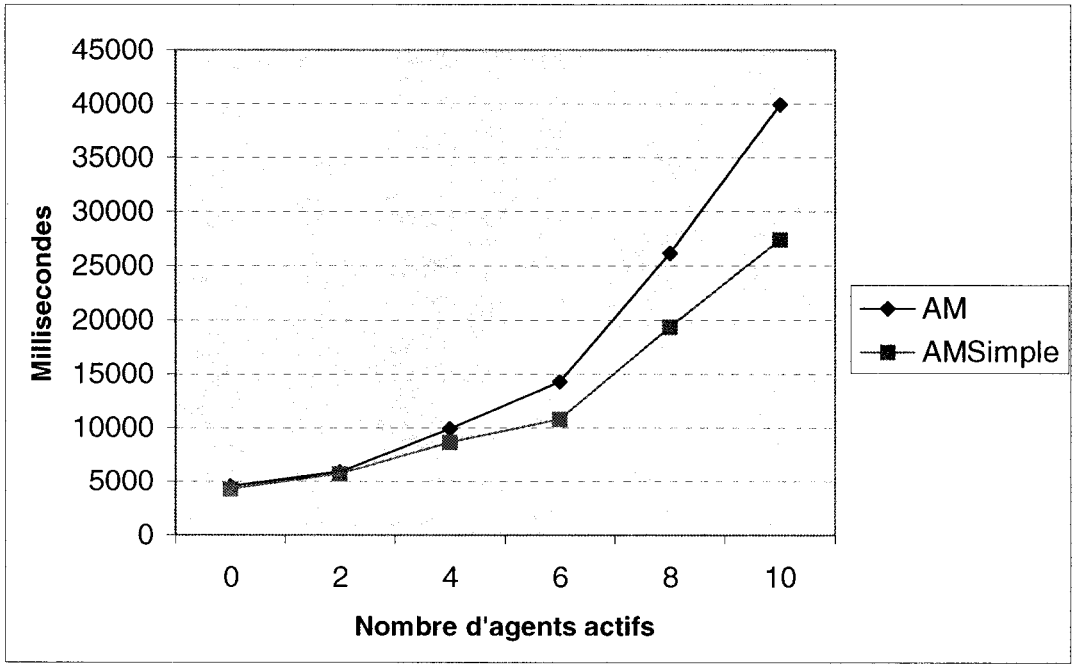

Figure 4.20 Impact du nombre des agents actifs sur le temps d'exécution

Nous concluons que l'ajout de notre protocole à une application d'agent mobile n'augmentera que légèrement le temps global d'exécution de l'application qui sera plus sécuritaire.

CHAPITRE V
CONCLUSION

Ce chapitre présente une synthèse des travaux que nous avons effectués sur la protection de l'agent mobile contre les attaques des plates-formes malveillantes. Nous verrons dans quelle mesure ces travaux ont été limités et par quels facteurs ils l'ont été. Enfin, nous donnerons des indications en vue de recherches futures.

5.1 Synthèse des travaux

Dans ce mémoire, nous avons présenté un protocole sécuritaire protégeant l'agent mobile contre les attaques des plates-formes, qui se base sur un agent sédentaire parfaitement coopérant s'exécutant à l'intérieur d'une tierce plate-forme fiable. L'idée de base était d'élargir les tâches de l'agent coopérant pour le rendre plus utile que dans les autres approches qui ont utilisé la notion des agents coopérants et qui l'ont restreint à l'enregistrement de l'itinéraire de l'agent mobile.

Dans une première phase, nous avons étudié d'une manière détaillée la vulnérabilité de l'agent mobile et nous avons recensé les différentes attaques possibles d'une plate-forme malveillante. Selon leur type, ces attaques ont été rassemblées dans des catégories qui ont été classées à leur tour suivant leur degré d'importance. Nous avons étudié les principales approches qui existent dans la littérature et nous avons constaté le manque d'un protocole complètement sécuritaire.

À partir de ce dernier constat, nous avons conçu notre protocole sécuritaire qui se base sur un agent sédentaire parfaitement coopérant. Le protocole a pour objectif de protéger une partie du code de l'agent mobile contre la modification et les exécutions incorrectes, de détecter les attaques contre les résultats intermédiaires obtenus à l'intérieur des plates-formes visitées, d'interdire la ré-exécution du code de l'agent mobile, de protéger celui-ci contre l'attaque appelée déni de service et d'enregistrer l'itinéraire de l'agent mobile afin de détecter les attaques liées à sa modification. Nous avons décrit les spécifications du protocole en précisant les tâches affectées à l'agent

coopérant, ainsi que les scénarios d'attaques que nous avons pu imaginer et nous avons expliqué la façon dont notre protocole permet de détecter ces attaques.

Après avoir spécifié le protocole, nous l'avons validé formellement afin de vérifier aussi bien ses propriétés générales que celles qui sont liées à la détection des attaques. Nous avons utilisé un model-checker pour modéliser le protocole et le simuler, et nous avons formalisé les propriétés à l'aide de la logique temporelle linéaire. La validation formelle nous a montré que notre protocole ne comporte aucune erreur de blocage ni de divergence. Nous avons pu expliquer quelques rares erreurs de validation lors de la vérification des propriétés liées à la détection des attaques. Les scénarios qui ont mené à ces erreurs de validation constituaient des comportements concordant avec les scénarios d'attaques prévus dans la phase de spécification.

Ensuite, nous avons implémenté notre protocole sur une plate-forme d'agents mobiles en prenant comme exemple un agent simple qui visite plusieurs plates-formes, exécute un code critique à l'intérieur de chaque plate-forme et retourne chez lui. Nous avons proposé et testé une méthode qui estime le compteur temporel *timeout* qui constitue l'idée principale de la détection de l'attaque de la ré-exécution du code de l'agent mobile. Cette dernière méthode nous a permis de détecter entre 90% et 100% de ce type d'attaque. Les tests de notre implémentation sur les différentes autres attaques ont révélé que l'agent sédentaire coopérant détecte presque toutes les attaques.

Enfin, dans l'objectif de mesurer les coûts et les performances de notre implémentation, nous avons choisi comme métrique le temps global d'exécution de l'agent mobile et le trafic généré par les différents agents. Nous avons comparé les résultats obtenus avec ceux d'un agent simple qui n'est pas sécuritaire. Les mesures ont démontré que l'ajout de notre protocole à une application d'agents mobiles n'aura pas une influence importante sur le trafic généré par l'application elle-même et il n'augmentera que légèrement le temps global d'exécution de celle-ci qui sera plus sécuritaire.

5.2 Limitations des travaux

La première limitation concerne l'environnement de validation offert par le model-checker SPIN. Celui-ci n'utilise que les combinateurs de la logique temporelle linéaire (LTL) pour formaliser les propriétés des systèmes à valider. Des combinateurs exprimant les états du passé ou bien les quantificateurs de chemins (i.e. quantifiant sur l'ensemble des séquences d'exécution), qui ne sont pas fournis par SPIN, pourront rendre les formules vérifiées plus robustes du point de vue de la validation formelle. Le manque de certains combinateurs était comblé par l'ajout des variables dans le modèle formel afin de formaliser les propriétés liées à la détection des attaques.

Les principales autres limitations de nos travaux concernent l'implémentation de notre protocole et les tests que nous avons effectués. Nous nous sommes heurtés à certains problèmes dont le manque de documentation précise sur le fonctionnement de la plate-forme et sur certaines librairies utilisées, principalement celle utilisée pour implémenter les compteurs temporels *timeout* et l'intervalle d'attente supplémentaire. Ce qui a rendu la tâche d'expliquer et de résoudre les erreurs de programmation très coûteuse en terme de temps de développement.

De plus que le manque d'une application pratique d'agents mobiles nécessitant la sécurité du code et des données a limité les résultats obtenus. En effet, nous avons voulu implémenter notre protocole sur une application de magasinage électronique dont le code source serait disponible. Mais le manque d'une telle application a restreint l'implémentation du protocole sur un exemple simple d'agent mobile qui nous a permis de tester l'efficacité et les coûts de nos algorithmes.

Enfin, les limitations sur les tests de notre implémentation viennent du fait que certains scénarios d'attaques imaginés sont impossibles sans modifier la façon dont la plate-forme exécute les agents mobiles. En effet, lors de ces attaques, la plate-forme ne se comporte pas d'une manière normale. Nous avons trouvé une solution à ce problème en modifiant le code de l'agent mobile pour chaque attaque de manière à ce que son comportement soit le même que s'il avait été attaqué par la plate-forme. De cette

manière, nous avons pu simuler les attaques et tester leur détection par l'agent sédentaire coopérant.

5.3 Indications de recherches futures

En ce qui concerne les recherches futures, il serait intéressant de tester le protocole sur une application réelle afin de confirmer les résultats que nous avons obtenus. En particulier, il serait très intéressant de refaire les études de l'impact de l'ajout de notre protocole à une application pratique sur le trafic généré et le temps d'exécution dans un environnement proche de celui de déploiement de cette application. Il faudrait prendre les mesures dans un environnement où les plates-formes exécutent un nombre aléatoire d'agents mobiles et où le réseau est relativement chargé.

Il faudrait développer aussi un mécanisme temps réel permettant le calcul du compteur de temps (*timeout*) utilisé pour détecter l'attaque de la ré-exécution. Ce mécanisme devrait être implémenté dans l'agent sédentaire coopérant et devrait prendre en considération les environnements d'exécution variables des plates-formes et du réseau. En effet, ces deux facteurs influencent directement le choix de la valeur initiale de ce compteur temporel. Ceci aurait pour objectif d'augmenter le taux où l'agent coopérant détecte ce type d'attaque et de diminuer le taux où il se trompe sur la fiabilité des plates-formes non-attaquantes.

L'approche que nous avons proposée est basée sur l'hypothèse stipulant l'existence d'une tierce plate-forme fiable. Celle-ci constitue le goulot d'étranglement de notre protocole et le point qui diminue sa tolérance aux pannes pour les raisons que nous avons expliquées dans les chapitres précédents. De ce fait, il faudrait trouver un mécanisme qui mettrait à la disposition des plates-formes plusieurs tierces plates-formes fiables, implémenterait des fonctionnalités de répartitions des demandes des plates-formes d'origine pour équilibrer les charges entre les différentes plates-formes, et permettrait à l'agent sédentaire coopérant de sauvegarder ses données d'une manière persistante afin de l'instancier correctement après une panne.

BIBLIOGRAPHIE

[ALG00] J. Algesheimer, C. Cachin, J. Cameniscsh, G. Karjoth, *Cryptographic Security for Mobile Code*, IBM Research Report, Zurich, Switzerland, Novembre 2000.

[ALL01] G. Allée, *Sécurité des Agents Mobiles : Protocole d'Enregistrement d'Itinéraire par Agents Coopérants*, Mémoire de maîtrise, École Polytechnique de Montréal, Canada, Février 2001.

[CHE95] D. Chess, B. Grosof, C. Harrison, D. Levine, and C. Paris, *Itinerant Agents for Mobile Computing*, Technical Report RC20010, IBM, Mars 1995.

[FAR96] W.M. Farmer, JD Guttman, and V Swarup, "Security for Mobile Agents : Issues and Requirements", in *Proceedings of the 19th Nationnal Information Systems Security Conference*, Octobre 1996, pp. 591-597.

[FAR96b] W. Farmer, J. Guttmann, V. Swarup, "Security for Mobile Agents : Authentication and State Appraisal", in *Proceedings of the 4th European symposium on research in computer security* (ESORICS), Springer-Verlag, Septembre 1996, pp. 118-130.

[FRE96] A. Freier, P. Karlton, P. Kocher, *The SSL Protocol, Version 3.0*, Internet Draft, Mars 1996.
http://home.netscape.com/eng/ssl3/ssl-toc.html

[HOL00] A. Holub, *Taming Java Threads*, Apress Publishers (Ed.), Chapters 5, 2000.

[HOL90] G.J. Holzmann, *Basic Spin Manual*, AT&T Bell Lab, Murray Hill, New Jersey, 1990.
http://spinroot.com/spin/Man/Manual.html

[HOH00] F. Hohl, "A Framework to Protect Mobile Agents by Using Reference States", in *Proceedings 20th International Conference on Distributed Computing Systems*, IEEE Computer Society, Los Alamitos (Ed.), California, 2000, pp. 410-417
ftp://ftp.informatik.uni-stuttgart.de/pub/library/ncstrl.ustuttgart_fi/TR-2000-03/TR-2000-03.ps.gz

[HOH98] F. Hohl, "Time Limited BlackBox Security : Protecting Mobile Agents From Malicious Hosts", in *Mobile Agents and Security,* LNCS 1419, Springer-Verlay, 1998, pp. 92-113.

[IAI02] Institute for Applied Information Processing and Communications, *Javadoc for IAIK-JCE 3.01*, Online Documentation, 2002.

http://jce.iaik.tugraz.at/products/01_jce/documentation/index.php

[KAR00] N.M. Karnik, A.R. Tripathi, "A Security Architecture for Mobile Agents in Ajanta", in *Proceedings 20th International Conference on Distributed Computing Systems*, Los Alamitos (Ed.), IEEE Computer Society, California, 2000, pp. 402-409.

[KOT99] D. Kotz, R. Gray, S. Nog, D. Rus, S. Chawla, G. Cybenko, "Agent TCL : Targeting the needs of Mobile Computers", in *Mobility : processes, computers, and agents,* Addison-Wesley (Ed.), 1999, pp. 514-523.

[KRO01] O. Krone, B. T. Messmer, H. Almiladi, T. Curran, "Java Framework for Negotiating Management Agents", in *Agent Technology for communication Infrastructure*, Wiley (Ed.), 2001, pp. 33-44.

[MIN96] Y. Minsky, R. Van Renesse, F. Scheinder, S. Stoller, "Cryptographic Support for Fault-Tolerant Distributed Computing", in *Proceeding of the 17th ACM SIGOPS*, European Workshop, 1996, pp. 109-114.

[PFL96] C.P. Pfleeger, *Security in computing, second edition*, Prentice Hall (Ed.), 1996, pp. 91-93.

[POS01] S.J. Poslad, R.A. Bourne, A.L.G. Hayzelden, P. Buckle, "Agent Technology for Communication Infrastructure : An Introduction", in *Agent Technology for communication Infrastructure,* Wiley (Ed), 2001, pp. 5-15.

[ROT98] V. Roth, "Mutual Protection of Co-Operating Agents", in *Secure Internet Programming*, Vitek et Jensen (Ed.), Springer-Verlag, Berlin, Allemagne, 1998, pp. 26-37.

[SAN98] T. Sander and C.F. Tschudin, "Protecting Mobile Agents Against Malicious Hosts, in *Mobile Agents and Security,* G. Vigna (Ed.), Lecture Notes in Computer Science, vol. 1419, 1998.

[SCH99] P. Schnoebelen, F. Laroussinie, M. Bidoit, B. Bérard, A. Petit, *Vérification de logiciels, techniques et outils du model-checking*, Vuibert (Ed.), Paris, France, 1999, pp. XIII-XV.

[VIG98] G. Vigna, "Cryptographic Traces for Mobile Agents, in *Mobile agents and security*, G. Vigna (Ed.), Springer-Verlag, 1998, pp. 137-153.

[WAL99] T. Walsh, N. Paciorek, D. Wong, "Security and reliability in Concordia", in *Mobility : processes, computers, and agents*, Addison-Wesley (Ed.), 1999, pp. 525-534.

[WIL99] U.G. Wilhelm, S. Staamann, and L. Buttyàn, "Introducing Trusted Third Parties to the Mobile Agent Paradigm", in *Cryptographic Security for Mobile Code*, IBM Research Report, 1999.

[YEE99] B. Yee, "A Sanctuary for Mobile Agents", in *Secure Internet Programming*, J. Vitek and C.D. Jensen (Ed.), *Lecture Notes in Computer Science*, vol. 1603, Springer, 1999, pp. 261-173.

[ZIM95] P.R. Zimmermann, *The Official PGP User's Guide*, MIT Press, 1995. http://mitpress.mit.edu/catalog/item/default.asp?ttype=2&tid=5518

ANNEXE I

CODE PROMELA DE MODÈLE FORMEL DE PROTOCOLE

Cette annexe contient le code Promela (SPIN) de modèle de notre protocole sécuritaire.

```
#define DeniDeService  0        /*Active l'attaque déni de service*/
#define ModifEntree   0         /*Active l'attaque modification du message Entrée*/
#define ModifSortie   0         /*Active l'attaque modification du message Sortie*/
#define CodeChange    0         /*Active l'attaque changement du code */
#define ExecutInc     0         /*Active l'attaque exécution incorrecte du code*/
#define ModifItin     1         /*Active l'attaque changement d'itinéraire*/

#define Id            byte      /* Identité d'une plate-forme */
#define Donnee        byte      /*Donnée transmise, soit dans le message Entrée soit dans le
                                  message Sortie*/
#define true          1
#define false         0
#define Rien          0
#define N             5         /* Le code critique modélisé par une fonction f(x)=x+N */
#define NPrim         6         /* La plate-forme attaquante change N par NPrim*/
#define MAX           150

#define TIMEOUT               timeout
#define Random                100

mtype = { Arrive, Entree, Sortie };  /* Le type de message échangé entre l'AM et l'AS*/

chan AMtoAS = [1] of {mtype, Id, Id, Donnee};  /* Modélise le canal de communication*/

Id Attaque;             /* L'identité d'une plate-forme qui attaque
                           Variable globale servant à vérifier des propriétés du protocole*/
Id Attaquante;          /*L'identité de la plate-forme que le système détecte son attaque*/
bool AttaqueEffectue;   /*False au début d'une exécution de l'AM, True si une Pi effectue une
                          attaque*/
bool AttaqueDetecte;    /* False au début d'une exécution de l'AM, True si l'AS détecte une
                          attaque*/
bool fin;               /*Détermine si l'AM est retourné à la plate-forme d'origine O*/

proctype AgentMobile()  /* Modélise le comportement de l'AM et celui d'une plate-forme
                           attaquante */
{
Id ID = 1;              /* L'identité de la plate-forme courante exécutant l'AM */
Id Prec = 0;            /* L'identité de la plate-forme d'où provient l'AM */
Id Suiv;                /* L'identité de la plate-forme où l'AM veut migrer */
bool CChange = false;   /* Reçoit la valeur True si une plate-forme attaquante
                           change le code de l'AM*/


Donnee X = 10;          /* Donnée en entrée */
Donnee R;               /* Résultat de la partie critique*/
```

```
fin = false;                /*Permet de vérifier une propriété LTL*/

progress:                   /*Le système doit toujours passé par ce point*/
do
::AttaqueDetecte = false;
 AttaqueEffectue = false;
 AMtoAS!Arrive(ID,Prec,Rien);
 if
#if DeniDeService
 ::Attaque = ID;            /*Attaque : déni de service, tue l'AM*/
  AttaqueEffectue = true;
  break
#endif
 ::if
  ::AMtoAS!Entree(ID,Rien,X)
#if ModifEntree
  :: AMtoAS!Entree(ID,Rien,X+10); /* Attaque : modification de message Entrée*/
    Attaque = ID;
    AttaqueEffectue = true;
#endif
  fi;
#if CodeChange
  if
  ::skip                        /* La plate-forme ne change pas le code */
  ::Attaque = ID;               /* Attaque : modification du code critique de l'AM*/
    AttaqueEffectue = true;
    CChange = true
  fi;
#endif
  if
  ::(CChange == false) ->       /* Simule la partie du code critique*/
                if
                ::R = X + N
#if ExecutInc
                ::Attaque = ID;   /* Attaque : exécution incorrecte du code*/
                  AttaqueEffectue = true;
                  R = X + NPrim
#endif
                fi
  ::else ->                      /* Le code a été changé par une plate-forme
                                    attaquante et demeure changé */
                R = X + NPrim
  fi;
  if                            /* Choix de la plate-forme suivante */
  ::Suiv = 0                    /* Choix de la plate-forme d'origine, fin d'itinéraire */
  ::Suiv = (ID + 1) % MAX;      /* Choix d'une autre plate-forme */
       if
       ::(Suiv == 0) -> Suiv = 1 /*Après le calcul de l'identité, affecte la valeur 1 au lieu de 0*/
       fi
#if ModifItin
  ::Attaque = ID;               /* Changement d'itinéraire avec modification de message
```

```
                                        Sortie*/
    AttaqueEffectue = true;
    Suiv = Random
#endif
  fi;
  if
  ::AMtoAS!Sortie(ID,Suiv,R)
#if ModifSortie
  ::Attaque = ID;                   /*Attaque : modification de message Sortie*/
    AttaqueEffectue = true;
    AMtoAS!Sortie(ID,Suiv,R+10)
#endif
#if DeniDeService
  ::Attaque = ID;                   /*Attaque déni de service , la plate-forme tue l'AM */
    AttaqueEffectue = true;
    break
#endif
  fi;
  if  /* Deplacement de l'AM */
  ::Suiv == 0 ->
          fin = true;
          break                     /*Retour a la plate-forme d'origine, fin d'itinéraire */
  ::else ->
          Prec = ID;
          if
          ::ID = Suiv               /*Déplacement vers une autre plate-forme  */
#if ModifItin
          ::Attaque = ID; /*Changement d'itinéraire sans modification de message Sortie */
            AttaqueEffectue = true;


            ID = Random

#endif
          fi
  fi
 fi
od
}

proctype AgentSédentaire()      /* Modélise le comportement de l'AS */
{
Id ID;                  /* L'identité de la plate-forme courante exécutant l'AM */
Id IDPrec;              /* L'identité de la plate-forme précédente communiqué par l'AM */
Id Prec = 0;            /* L'identité de la plate-forme précédente enregistré par l'AS*/
Id Suiv = 1;            /* L'identité de la plate-forme suivante où l'agent veut migrer */


Donnee X,R,RAS;
end1 :          /* Après l'exécution normale d'un AM, l'AS attend de communiquer avec O*/
do
::AMtoAS?Arrive(ID,IDPrec,Rien);        /*Réception de message Arrive*/
```

```
if                                      /*Vérification de l'itinéraire */
::(ID == Suiv) && (IDPrec == Prec) ->
        printf("Itinéraire correct");
        Prec = ID                       /*Enregistrement de l'itinéraire*/
::else ->
        Attaquante = Prec;              /*La plate-forme précédente qui a changé l'itinéraire*/
        printf("Itinéraire modifié");
        AttaqueDetecte = true;
        assert(Attaque == Attaquante);
        break
fi;
if
::AMtoAS?Entree(ID,Rien,X);  /* Réception de message Entrée*/
::TIMEOUT ->                 /* Expiration de délai d'attente */
          Attaquante = ID;   /* Détection de déni de service*/
          AttaqueDetecte = true;
          assert(Attaque == Attaquante);
          break              /* Re-initialisation du protocole */
fi;
RAS = X + N;                 /* Exécution de la partie du code critique*/
if
::AMtoAS?Sortie(ID,Suiv,R) -> /* Réception de message Sortie */
                              if
                              ::skip  /* La réception se fait à l'intérieur de Timeout du protocole*/
#if DeniDeService
                              ::printf("PLate-forme attaquante");  /* La réception se fait après
                                                                    l'expiration de Timeout du
                                                                    protocole*/
                                Attaquante = ID; /* L'instruction assert est omise, vu que cette partie
                                                    est simulée seulement dans l'AS*/
                                AttaqueDetecte = true;
                                goto end
#endif
                              fi
::TIMEOUT ->                  /* Expiration de délai d'attente */
          Attaquante = ID;    /* Attaque détectée déni de service*/
          AttaqueEffectue = true;
          AttaqueDetecte = true;
          assert(Attaque == Attaquante);
          break               /* Re-initialisation du protocole */
fi;
if                            /* Calcul et comparaison des résultats */
::(RAS == R) ->
        printf("Exécution correcte")
::else ->
        printf("Plate-forme attaquante");
        Attaquante = ID;
        AttaqueDetecte = true;
        assert(Attaque == Attaquante) /*Vérifie que le protocole détecte l'attaque de la plate-
                                        forme attendue*/
fi;
```

```
end: skip;
od
}


init   /* Lance les deux processus AgentMobile Et AgentSédentaire*/
{ atomic
  {
          run AgentMobile(); run AgentSédentaire()
  }
}
```